PB-216 729

HYDRO REFERENCE MANUAL - A CONTENT-ORIENTED COMPUTER LANGUAGE FOR HYDRAULIC ENGINEERING

William D. McNally

Carnegie Institute of Technology
Pittsburgh, Pennsylvania

September 1966

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

WP-00415

PB 216 729

HYDRO
REFERENCE MANUAL

A CONTENT–ORIENTED
COMPUTER LANGUAGE FOR
HYDRAULIC ENGINEERING

BY

WILLIAM D. McNALLY

CARNEGIE INSTITUTE OF TECHNOLOGY
DEPARTMENT OF CIVIL ENGINEERING
SEPTEMBER 1966

Contract WP-00415, U.S. Public Health Service

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

294

MEMORANDUM

CARNEGIE INSTITUTE OF TECHNOLOGY
PITTSBURGH, PENNSYLVANIA 15213

SUBJECT: HYDRO

This is the HYDRO Reference Manual, describing in detail the way in which HYDRO is implemented on an existing computer system.

A companion manual, the HYDRO Users Manual, giving detailed instructions for the use of the language, and describing in detail each procedure, is also available.

We will greatly appreciate any comments you may have on the enclosed manual, since feedback from potential users, as well as a more general debate on the effectiveness of this type of tool, is vital to the future development of the concepts embodied in HYDRO.

We shall be glad to provide all the assistance we can to anyone interested in using HYDRO.

G Bugliarello

George Bugliarello
Professor of Biotechnology and
Civil Engineering,
Chairman, Biotechnology Committee.

PREFACE

HYDRO is a content-oriented computer language for the solution of hydraulic engineering problems. Its underlying philosophy and the method for putting an existing HYDRO system to use are described in the HYDRO Users Manual (8), a companion publication to this one. That manual should be well understood and its contents put into practice prior to the reading of this manual.

A complete description of the HYDRO compiler is presented in this manual. This description is intended for use by anyone wishing to implement his own HYDRO system, or for those in charge of the care and expansion of an existing system.

An entire system is discussed herein, with regard to the elements which comprise it and their relative operation when a HYDRO program is run. The HYDRO compiler, as well as the procedures which read data in HYDRO format, are described by means of fully explained flow charts. The augmentation of HYDRO through the design and insertion of new procedures is also explained. Finally, aspects of the system peculiar to Carnegie Tech are discussed so that the reader can relate them to similar facilities at his own computation center.

ACKNOWLEDGEMENTS

Work on HYDRO was initiated in 1962 and has been supported since that time under a grant from the United States Public Health Service.

The design of a compiler to execute input HYDRO programs was begun while the author was a graduate student and was advanced to its present stage as a part of post-doctoral work. This development was completed under the guidance of Dr. George Bugliarello of the Civil Engineering Department of Carnegie Institute of Technology.

The author is also grateful for the assistance of David Thomas of the Physics Department of Carnegie Tech, who originally designed many of the procedures incorporated into the HYDRO compiler.

TABLE OF CONTENTS

Page

Preface ii

Acknowledgements iii

Table of Contents iv

1. INTRODUCTION 1.1

2. THE HYDRO SYSTEM 2.1

  2.1 Components of the System 2.1.1

  2.2 Operation of the System 2.2.1

3. GENERAL SYSTEM FLOW CHART 3.1

  3.1 Flow Chart 3.1.1

  3.2 Explanation of the Flow Chart 3.2.1

4. COMPILER PROCEDURES AND VARIABLES 4.1.1

  4.1 Procedures 4.1.1

  4.2 Variables 4.2.1

5. DETAILED COMPILER FLOW CHART 5.1

  5.1 Flow Chart 5.1.1.1

    5.1.1 Phase One - Recognition 5.1.1.1

    5.1.2 Phase Two - Assembly 5.1.2.1

    5.1.3 Phase Three - Transferral 5.1.3.1

  5.2 Explanation of the Flow Chart 5.2.1.1

    5.2.1 Phase One - Recognition 5.2.1.1

    5.2.2 Phase Two - Assembly 5.2.2.1

    5.2.3 Phase Three - Transferral 5.2.3.1

6. DATA READING PROCEDURES AND VARIABLES 6.1.1

  6.1 Procedures 6.1.1

  6.2 Variables 6.2.1

7. DETAILED DATA READ FLOW CHART 7.1

7.1 Flow Chart 7.1.1

7.2 Explanation of the Flow Chart 7.2.1

8. AUGMENTATION OF HYDRO 8.1

8.1 Rules Governing HYDRO Variables 8.1.1

8.2 Design of Procedures for Insertion into HYDRO 8.2.1

8.3 Insertion of a New Procedure into HYDRO 8.3.1

9. IMPLEMENTATION OF HYDRO ON OTHER COMPUTING FACILITIES 9.1

Bibliography B.1

Appendix I - Relation of HYDRO to Carnegie Tech Computing Facilities A1.1

Appendix II - The Segments of the HYDRO Compiler A2.1

Appendix III - The Seven HYDRO Data Read Procedures A3.1

1. INTRODUCTION

The HYDRO computer system, for the solution of hydrology and hydraulic engineering problems, consists of an input language for the user and a processor for accepting and translating inputs into an equivalent program for execution. The input language is presented in the HYDRO Users Manual (8). The present manual describes the HYDRO processor in detail as well as the means for augmenting it and implementing it at different computer centers.

Chapter Two gives an introduction to the processing system by describing its software components and the general manner in which they interact during the operations of translation and execution of a HYDRO program. A general flow chart of the HYDRO compiler is presented and explained in Chapter Three; used in conjunction with Chapter Two it gives an overall view of how the compiler functions.

Chapters Four and Five treat the compiler in detail. In Chapter Four each of the procedures and variables used within the compiler is identified and explained briefly. Then in Chapter Five a detailed compiler flow chart is presented and explained in terms of the actual variables used in the compiler programs. This flow chart should be studied in conjunction with the actual compiler program listings given in Appendix II for a full understanding of the recognition, assembly, and execution processes of the compiler.

Chapters Six and Seven give detailed explanations of the seven procedures used for reading data in the HYDRO format during the execution of ALGOL programs assembled from input HYDRO code. These procedures are placed at the head of each program assembled from HYDRO and are then used during the execution of the resulting program to read data from copies of the original HYDRO data cards of the input program. In Chapter Six the procedures and variables used for the reading of data are introduced and explained. Chapter Seven presents a detailed flow chart of the seven data reading procedures, followed by an explanation. Appendix III contains the computer listings of these procedures, and should be studied in conjunction with Chapters Six and Seven. The above four Chapters, Four to Seven, must be understood by anyone wishing to alter HYDRO for use on his own computing system, as well as by those maintaining existing HYDRO systems.

Chapters Eight and Nine deal with the problems of augmenting an existing HYDRO system through the addition of a new procedure, and of implementing HYDRO at other computing facilities. The first Appendix assists in the understanding of this material since it describes the features of the existing HYDRO system which are peculiar to Carnegie Tech.

2 THE HYDRO SYSTEM

The HYDRO system has two principal components: a compiler and a procedure library. When a HYDRO program is run, the compiler translates it and assembles from the library an equivalent ALGOL program which is then executed.

Two additional components of lesser importance are also involved with the HYDRO system. The first is a general workspace in which the ALGOL program, which is the equivalent of the input HYDRO program, is assembled, and from which it runs. The second is a copy of the entire compiler, stored on computer records, and used only when additions or alterations are being made to the system.

Thus, the HYDRO system has four components - the first three used during the operation of HYDRO, and the fourth used during the alteration of HYDRO:

1. The actual "compiler in use," broken into several smaller parts due to size limitations in the computer, and stored onto the so-called "segments" used when a HYDRO program is submitted for execution,

2. The language library, containing all the ALGOL code necessary for assembling equivalent programs which execute the tasks called for by HYDRO commands, and herein called AND RECORD TWO,

3. A large workspace on which the equivalent ALGOL programs are assembled, called the AND SCRATCH AREA, and

4. A copy of the working compiler, called the "compiler in storage," and herein referred to as AND RECORD ONE.

These components, indicated schematically on Figure 1, are all located on peripheral storage (disc file) at the Carnegie Tech Computation Center. They could just as easily be located on magnetic tapes.

This chapter discusses these four components and the purpose of each. In the latter section of the chapter their interaction during the running of a HYDRO program is outlined. The size of the various parts of the HYDRO system in terms of the computer storage required for each is also presented.

2.1 Components of the System

The Compiler in Use

The first component of the system is the working compiler. It is presently divided into seven sub-programs which are stored on segments of a disc file. It has three principal Phases to its execution -- Phase One requiring two segments, Phase Two using four segments, and Phase Three using only one segment. Each of these segments is a complete program in itself, but they all work together as one long ALGOL program which is executed each time a HYDRO program is submitted. (More information on the use of segmented programs is given in Appendix I).

The compiler is divided into seven segments only because of the inability of the available core storage inside the computer to handle a program the size of the entire compiler. There is no reason why the seven segments of the compiler could not be merged into one program if more space was available.

Each segment of the compiler is coded as an ALGOL procedure which is called at the end of the segment. This is done in order to set the segments apart and show that they are capable of working independently of each other. The two Phase One segments contain the procedure PHASE.ONE.RECOGNIZE. Each of the Phase Two segments has the procedure PHASE.TWO.ASSEMBLE. The principal procedure of Phase Three, the final segment, is PHASE.THREE.TRANSFER.

On the average, these segments range in length from 600 to 800 card images. When executed as programs, they average about 12,000 to 14,000

THE HYDRO SYSTEM			
COMPILER IN USE	LANGUAGE LIBRARY	ASSEMBLY AREA	COMPILER IN STORAGE
SEGMENTS	AND RECORD TWO	AND SCRATCH AREA	AND RECORD ONE
DISC-TAPE	DISC-TAPE	DISC-TAPE	DISC-TAPE
SEVEN SECTIONS CODED IN ALGOL ARRANGED FOR USE WHEN A HYDRO PROGRAM IS SUBMITTED	HYDRO SUBROUTINES CODED IN ALGOL MATERIAL IS DRAWN FROM VARIOUS PARTS OF THE LIBRARY WHEN A HYDRO PROGRAM IS RUN	GENERAL WORK-SPACE USED FOR ASSEMBLY OF ALGOL PROGRAM FROM HYDRO CODE	SEVEN SECTIONS CODED IN ALGOL USED ONLY WHEN MAKING ALTERATIONS TO THE HYDRO SYSTEM NOT WHEN RUNNING A HYDRO PROGRAM

FIG. 1 COMPONENTS OF THE HYDRO SYSTEM

words each of internal computer space for storage of the compiled program and required data. To this must be added the space required for the ALGOL translator which compiles them (12,000 words) as well as the monitor for the entire computer (8,000 words). In brief, these programs, executed individually, require most of the capacity of the 32,000 word computer on which they are executed.

When a HYDRO program is submitted, the first segment of Phase One is automatically executed. Along with the other Phase One segment, it recognizes the input code of the HYDRO program, and arranges a list of numbers within the computer which represent that code. Links are then made to the segments comprising Phase Two, and they are loaded into the computer for execution. These operate on the stack of numbers compiled by Phase One, and set up a list of number pairs representing blocks of ALGOL code to be withdrawn from the language library in order to assemble an ALGOL program which is the equivalent of the inputted HYDRO code. The Phase Three segment of the compiler then transfers these blocks of code to an assembly area and executes the resulting program.

The Language Library

The second important component of the system is the language library, AND RECORD TWO. This "record," or "file," also on disc storage, is located within the Carnegie Tech AND system so that it can be easily altered and augmented. (The AND system is explained more fully in Appendix I).

The entire code on AND RECORD TWO, as opposed to simply the HYDRO procedure calls, is what is referred to as the "language." This involves

ALGOL code for all of the HYDRO procedures and procedure calls, the single variable and array read cards, the cards declaring all variables, and the data reading procedures. Although the data read procedures are not called by the HYDRO user, they are considered as part of the "language" since they are assembled and run within the final ALGOL program assembled from the procedures called in HYDRO. They are not executed during the HYDRO compilation process, but at run time of the final program assembled from HYDRO.

The data read procedures are the first material located on AND RECORD TWO, and are automatically assembled by the compiler at the top of every ALGOL program constructed on the SCRATCH AREA. These procedures are required for the reading of data from the HYDRO data cards assembled at the end of the ALGOL program, for these cards are not read prior to the execution of this final ALGOL program. The seven procedures involved are NEXT.COLUMN, NUMERICAL.DATA, ALPHA.NUMERICAL,DATA, DEFINE.SUBSCRIPTS, DATA.INPUT, SINGLE.READ, and ARRAY.READ.

Following the data read procedures is a short section of error codes executed when mistakes are encountered during the reading of input data.

Following the error codes are the declarations of all single variables used in any of the procedures which can be called in the HYDRO language. These declarations are transferred as a block into each ALGOL program assembled from HYDRO, since the number of storage locations involved is relatively small.

Next on RECORD TWO are the cards which read data into all the single read-in variables which can be used in the various HYDRO procedures. These all contain calls on SINGLE.READ. Whenever data are given in a HYDRO program for one of these read-in variables, the appropriate data read card is transferred from this section of RECORD TWO into the ALGOL program being developed.

Following the single read cards are the sections of data read code for each of the read-in arrays which can be used in the HYDRO procedures. When data are entered in a HYDRO program for one of these arrays, the appropriate section of array read code is transferred from this section of RECORD TWO to the program being developed in the SCRATCH AREA. Each of these sections of code contains the array declaration plus a call on the ARRAY.READ procedure.

After the array read cards on RECORD TWO come the sections of code representing material which must be assembled global to any of the procedures. This code consists principally of array declarations and of simple calculations of array bounds which will enable some array declarations to be made.

The final section of code on AND RECORD TWO, and by far the largest, contains the bodies of all procedures it is possible to call in HYDRO.

The language library is stored on the disc file as card images. The entire record at the present time, with about 80 or 90 procedures of all sizes available to the user, occupies approximately 4000 card images of space.

The Assembly Area

On the assembly area, the ALGOL program drawn from the library (AND RECORD TWO) which represents the input program of the HYDRO user, is developed. This workspace is a special file of the Carnegie Tech AND system, called the AND SCRATCH AREA.

This file is not an exclusive part of the HYDRO system. It is available to everyone for use during the execution of programs, as a logical file for the temporary storage and assembly of other programs. During Phase Three of the HYDRO compiler, sections of code are transferred from AND RECORD TWO to the SCRATCH AREA to make up an ALGOL program. During Phase One of compilation each input data card from the HYDRO program is stored near the end of the SCRATCH AREA as it is picked up. These card images remain there undisturbed throughout the processing of Phase Two and at the end of Phase Three are transferred forward on the SCRATCH AREA to the end of the ALGOL program assembled there. The resulting program is then read into the computer core and executed.

Like the other parts of the HYDRO system, the assembly area is located on disc. Approximately four thousand card images, the same length as AND RECORD TWO, would be adequate for the assembly of any program submitted in HYDRO. The SCRATCH AREA at Carnegie Tech, however, is many times longer than this, and is never completely used.

The Compiler in Storage

The final component of the system is AND RECORD ONE. On AND RECORD ONE a replica is maintained in storage of the actual compiler in use. This is simply a back to back listing of the segmented compiler programs. The sole purpose of this record is for use in altering or making additions to the compiler. By means of the AND language explained in Appendix I, these additions can easily be made to the programs on AND RECORD ONE and the revised programs stored back onto this AND RECORD. These sections of the compiler are then copied from this AND RECORD back onto the segments of the "compiler in use" for operation during the running of HYIRO programs.

The size of AND RECORD ONE is simply the summation of the card images of the compiler segments. There are presently seven segments of 600 cards each, resulting in AND RECORD ONE being approximately 4000 card images in length.

2.2 Operation of the System

When a HYDRO program is submitted and run, the various components of the operating system - the compiler segments, the library, and the assembly area - interact with one another to produce the result intended by the user. The order and the manner in which these components operate is described below and illustrated in Figures 2, 3, and 4.

Phase One

The operations of Phase One are illustrated in Figure 2.

When a HYDRO program is submitted, the computer automatically LINKs to the first segment of the compiler, which is the initial part of Phase One. This causes the contents of the segment to be dumped from disc storage into the computer core and run as an ALGOL program. This ALGOL program uses the cards of the HYDRO input program as its data.

The purpose of Phase One is to alphanumerically read each of the HYDRO input cards containing either procedure calls or data, and to scan them. By means of special procedure calls it then recognizes the names of the commands and data entered, and stores an identifying number for each in a one-dimensional array called STACK. This list of numbers can then be used for processing in Phase Two, since the computer handles numbers much more conveniently than it does alphanumeric strings. Phase One also locates most of the errors it is possible for a user to make in HYDRO, and for each that it encounters it prints a diagnostic message to the user.

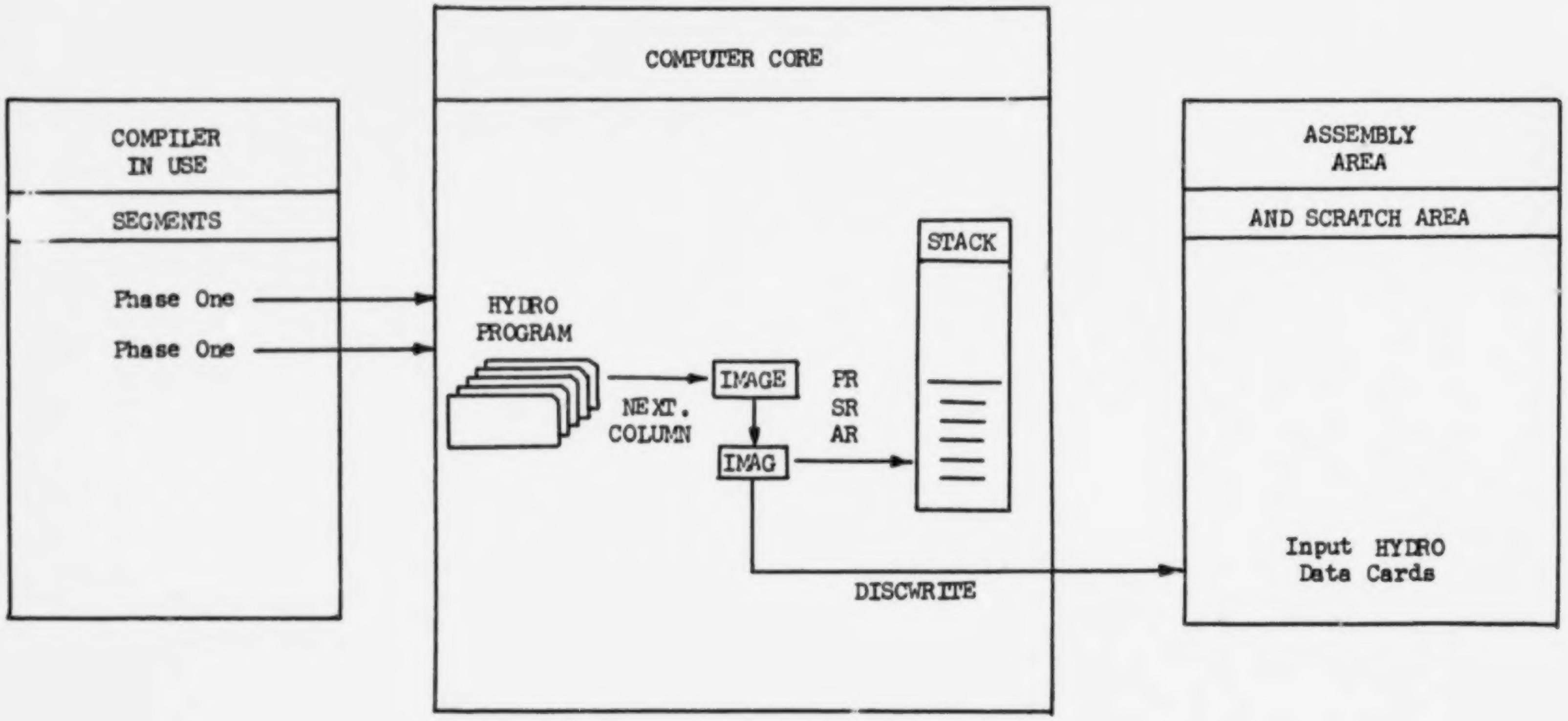

COMPILER
IN USE
SEGMENTS
Phase One
Phase One
COMPUTER CORE
HYDRO
PROGRAM
NEXT.
COLUMN
IMAGE
IMAG
PR
SR
AR
STACK
DISCWRITE
ASSEMBLY
AREA
AND SCRATCH AREA
Input HYDRO
Data Cards

FIG. 2 OPERATION OF PHASE ONE

As soon as Phase One is entered four procedures used by it, NEXT. COLUMN, PR, SR, and AR are declared, as well as the Computer Center library routines of LINK and DISCWRITE (See Appendix I). PR, SR, and AR represent "procedure recognition," "single variable recognition," and "array recognition" respectively.

Phase One scans the input cards of the HYDRO program one by one, storing them by means of the NEXT.COLUMN procedure, into an 80-unit logic vector called IMAGE. This procedure then moves a pointer across the IMAGE vector column by column. When the end of the vector (or card) is reached, it stores the following input card into IMAGE and begins scanning at column one again.

When the Phase One program discerns that a procedure call or a data variable lies within the IMAGE vector, it stores that call or variable into a smaller logic vector called IMAG. If the IMAG vector contains a procedure call, the procedure PR is called for all possible procedure calls beginning with the same letter as the IMAG vector. When one is recognized which coincides with the contents of IMAG, its identifying number is stored into the STACK. A similar process is followed when single variables and arrays are scanned and stored into IMAG, using SR and AR instead of PR. The identifying numbers for single variables begin at zero, those for arrays at 10,000, and those for procedures at 20,000.

During the operation of Phase One, LINKs are often made back and forth between the two segments which comprise it, since calls on the recognition procedures for the HYDRO procedure calls and data variables

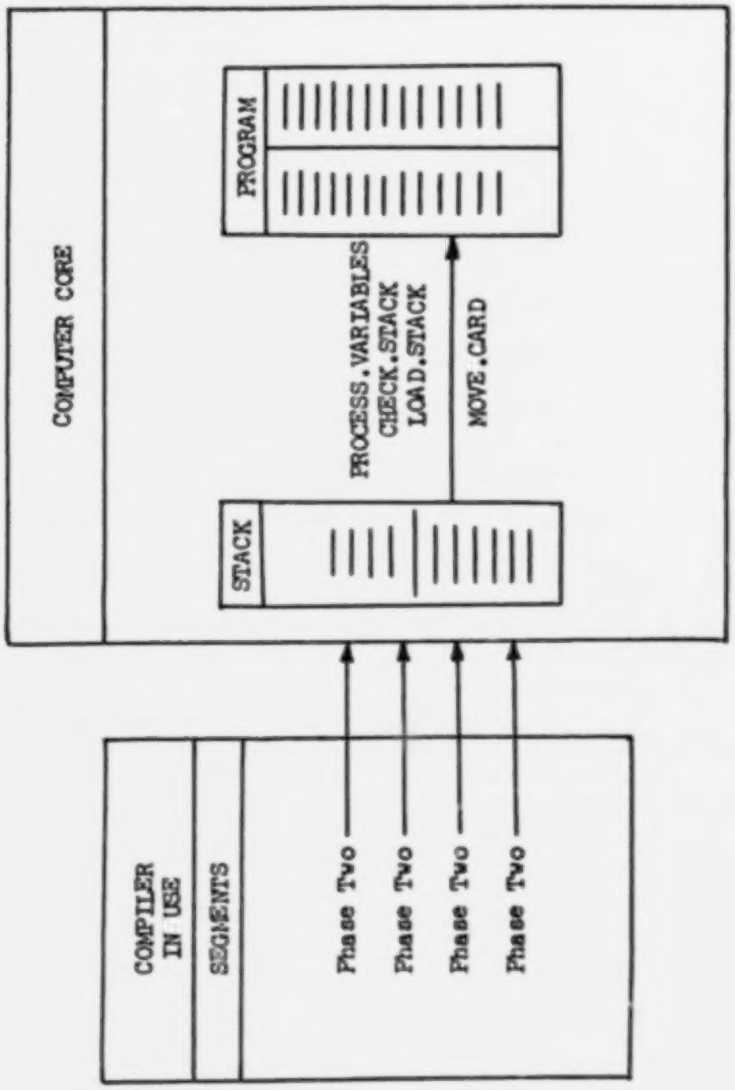

COMPUTER CORE
PROGRAM
STACK
PROCESS.VARIABLES
CHECK.STACK
LOAD.STACK
MOVE.CARD
COMPILER IN USE
SEGMENTS
Phase Two
Phase Two
Phase Two
Phase Two

FIG. 3 OPERATION OF PHASE TWO

are distributed throughout both segments. When this occurs, the segment LINKed to replaces the segment which called for it and begins operating in core as an ALGOL program.

Phase One also stores the images of all HYDRO data cards in the input program onto the end of the AND SCRATCH AREA. The DISCWRITE routine is used to do this. These cards are retained there throughout the other Phases of compilation, and at the end of Phase Three are moved forward to become data for the ALGOL program which Phase Three assembles on the front portion of the SCRATCH AREA.

Phase Two

Phase Two operations are illustrated on Figure 3.

At the completion of Phase One, a LINK is made to the first segment of Phase Two. It then replaces Phase One in core and commences operation.

The procedures employed by Phase Two are the following: MOVE.CARD, IN.THE.STACK, CHECK.STACK, LOAD.STACK, SINGLE.VARIABLE, ARRAY.VARIABLE, TEST.BOUNDS, PROCESS.VARIABLES, PROCESS.VARIABLES.EXCEPT, and PROCESS.VARIABLES.UNTIL.

Within Phase Two there is a section of processing code for each of the procedure numbers which may have been placed into the STACK by Phase One. As the STACK is processed by Phase Two, these sections of code are executed in a sequence determined by the order of procedure call numbers in the STACK. The principal procedures of PROCESS.VARIABLES, CHECK.STACK, and LOAD.STACK are called to process the data variable STACK numbers following each procedure number, and then MOVE.CARD is called to assemble

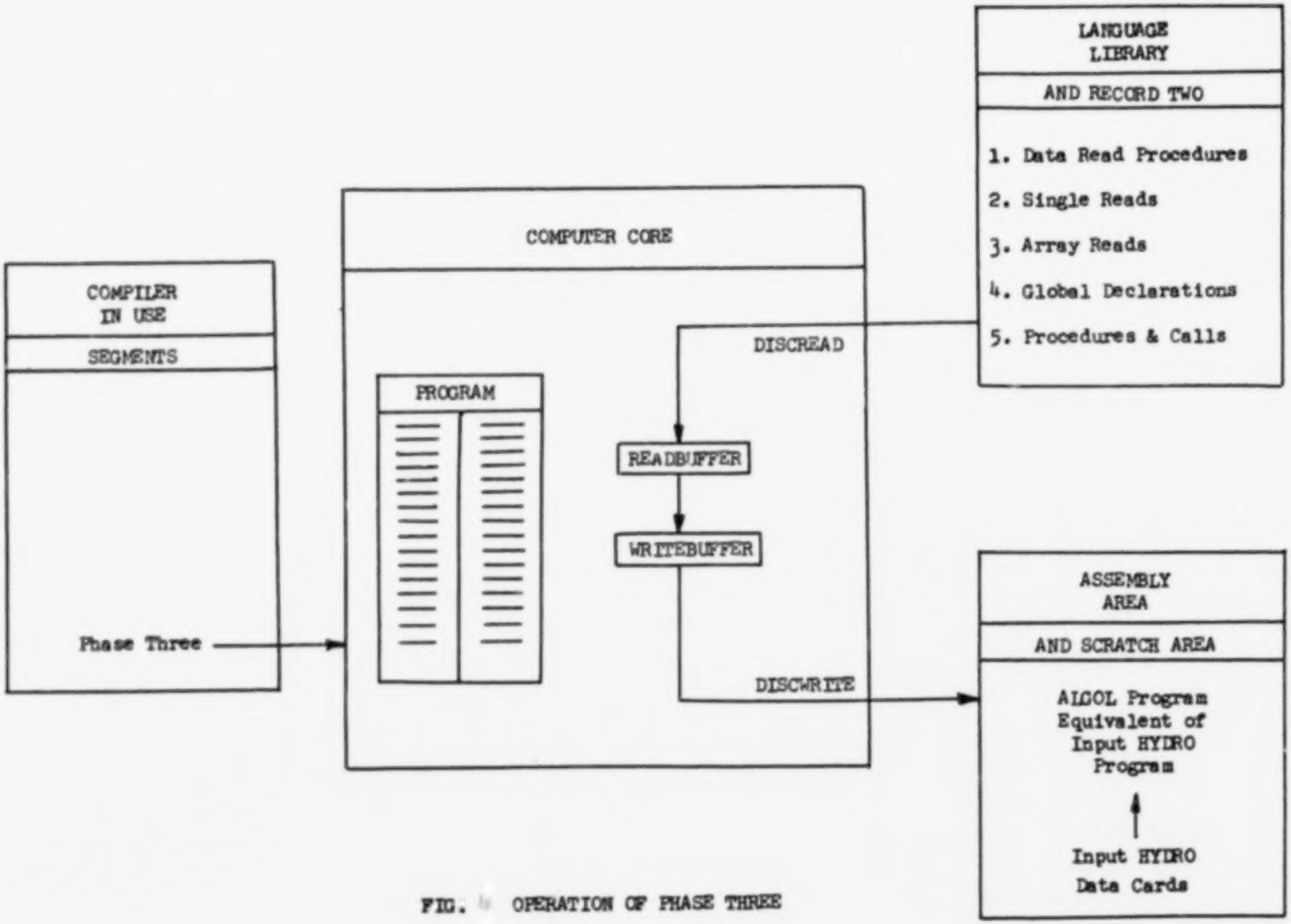

COMPILER IN USE
SEGMENTS
Phase Three
COMPUTER CORE
PROGRAM
DISCREAD
READBUFFER
WRITEBUFFER
DISCWRITE
LANGUAGE LIBRARY
AND RECORD TWO
1. Data Read Procedures
2. Single Reads
3. Array Reads
4. Global Declarations
5. Procedures & Calls
ASSEMBLY AREA
AND SCRATCH AREA
ALGOL Program Equivalent of Input HYDRO Program
Input HYDRO Data Cards

FIG. 4 OPERATION OF PHASE THREE

a new set of number pairs into an array called PROGRAM. These pairs of numbers represent the first and last card images of blocks of ALGOL code on AND RECORD TWO. These blocks will be transferred by Phase Three onto the SCRATCH AREA to form the ALGOL program called for by the initial HYDRO code.

As in Phase One, Phase Two LINKs back and forth among the segments which comprise it, depending upon which procedures have been called by the user. These segments all perform similar tasks, but for different procedures of the HYDRO language.

Phase Three

The operations of Phase Three are illustrated in Figure 3.

Following Phase Two, when the STACK has been completely scanned and the PROGRAM array has been set up for the ALGOL program to be assembled, a LINK is made to the Phase Three segment.

The purpose of Phase Three is to transfer the code represented by PROGRAM from AND RECORD TWO to the SCRATCH AREA, place the data card images from the end of the SCRATCH AREA behind this program, and execute the result. There are two procedures employed by Phase Three to do this: TRANSFER.CARD and TRANSFER.BUFFER. These two procedures also make use of the Computation Center library routines, DISCREAD and DISCWRITE. (See Appendix I).

As Phase Three operates, it transfers large amounts of code from the language library, AND RECORD TWO, to the assembly area, the AND SCRATCH AREA. The number pairs of the PROGRAM array tell the TRANSFER procedures which pieces of code are to be withdrawn from the library.

These blocks of code may be the data read procedures, calls on these procedures, variable declarations, or procedure bodies.

Phase Three calls the TRANSFER.CARD procedure for each pair of numbers in the PROGRAM array. This causes equal sized blocks of code to be withdrawn by the DISCREAD routine from AND RECORD TWO. These are placed into an array in computer core called READBUFFER. This buffer is then scanned, and the code located within it which is required as part of the ALGOL program begin assembled onto the SCRATCH AREA is transferred to the WRITEBUFFER by means of TRANSFER.BUFFER. When WRITEBUFFER fills with code it is written by DISCWRITE onto the AND SCRATCH AREA. This transferral process continues until the PROGRAM array is completely processed. At this point, the ALGOL program on the SCRATCH AREA represents the HYDRO program submitted by the user. The data in temporary storage at the end of the SCRATCH AREA are then transferred forward to the end of the code just assembled, and the complete result is put into the computer core and executed. This completes compilation and execution of the original HYDRO program.

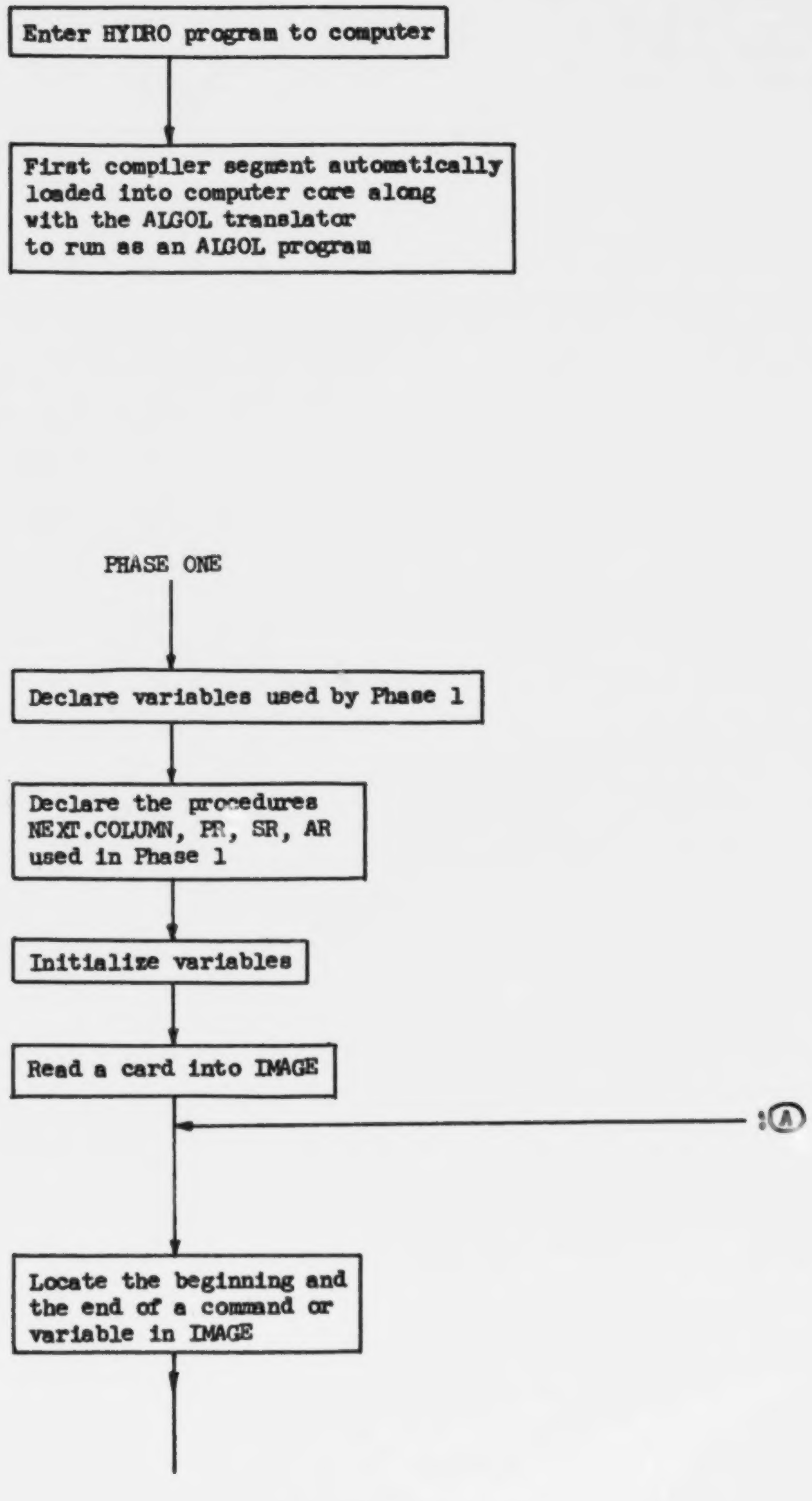
3. GENERAL SYSTEM FLOW CHART

This Chapter contains a general flow chart of the entire HYDRO system, followed by its explanation. The majority of this flow chart is involved with the compiler; another flow chart explaining the compiler in detail is given in Chapter 5. A small section at the end of the flow chart is involved with the execution of the ALGOL program assembled from the input HYDRO code. The primary items requiring explanation here are the seven data reading procedures which are placed at the head of every ALGOL program assembled from HYDRO. A detailed flow chart is given for these in Chapter 7.

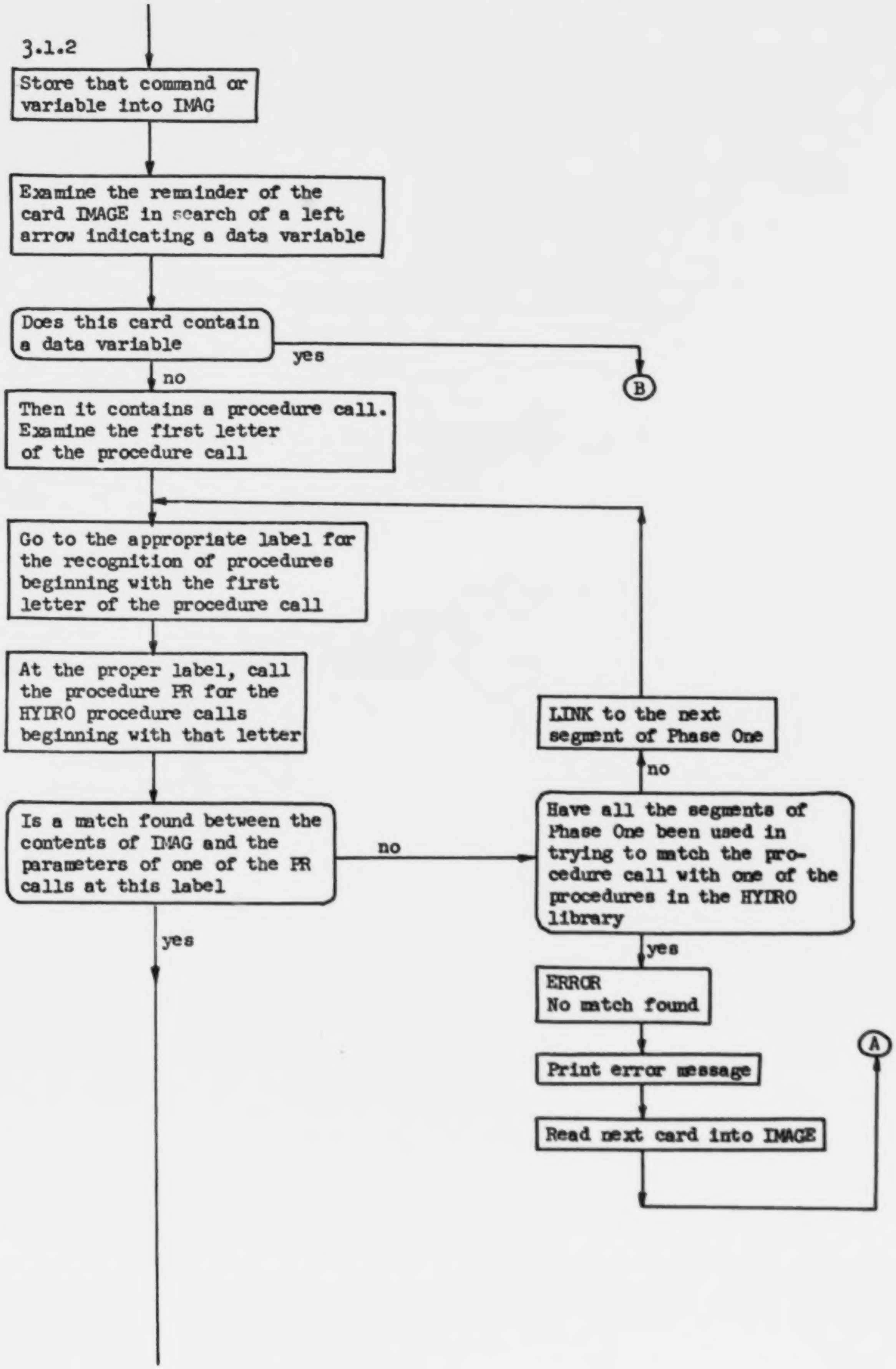
The purpose of the flow chart given here is to generate in the reader additional familiarity with the compilation process before examining it in detail. Thus steps in this chart represent entire blocks of code in the actual compiler programs.

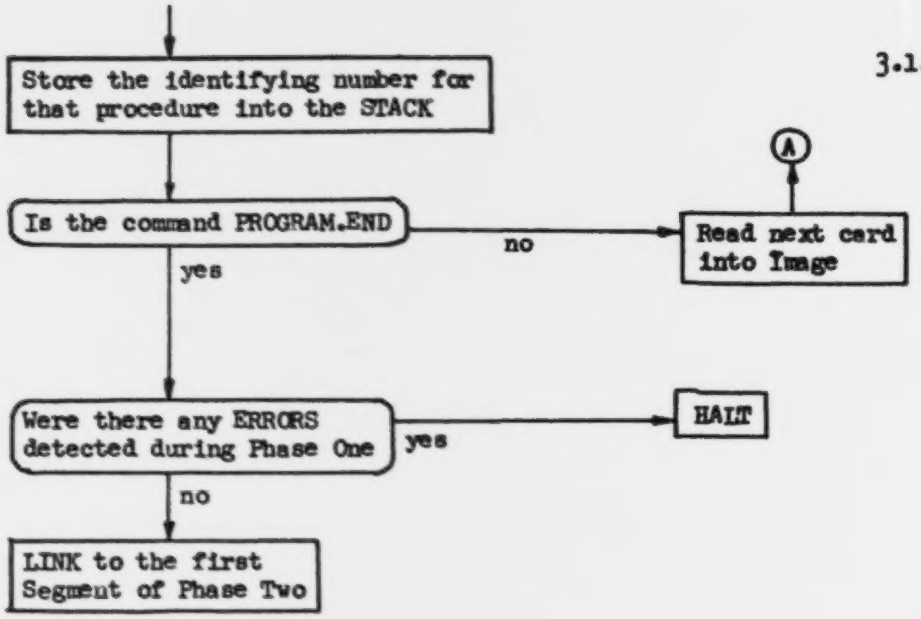
In the flow charts of this and the following chapters a rectangular box around a statement indicates that it is an assignment statement or a group of them, or a call on a procedure declared previously, or a small iteration loop. Rectangular boxes with rounded corners, on the other hand, represent decision points. Each of these will have both a YES and a NO exit. Small circled letters and numbers followed by double dots (: or ..) represent labels. Arrows indicate the direction of flow. In all of the flow charts, arrows drawn to the bottom of a page indicate that the chart continues at the same position at the top of the following page.

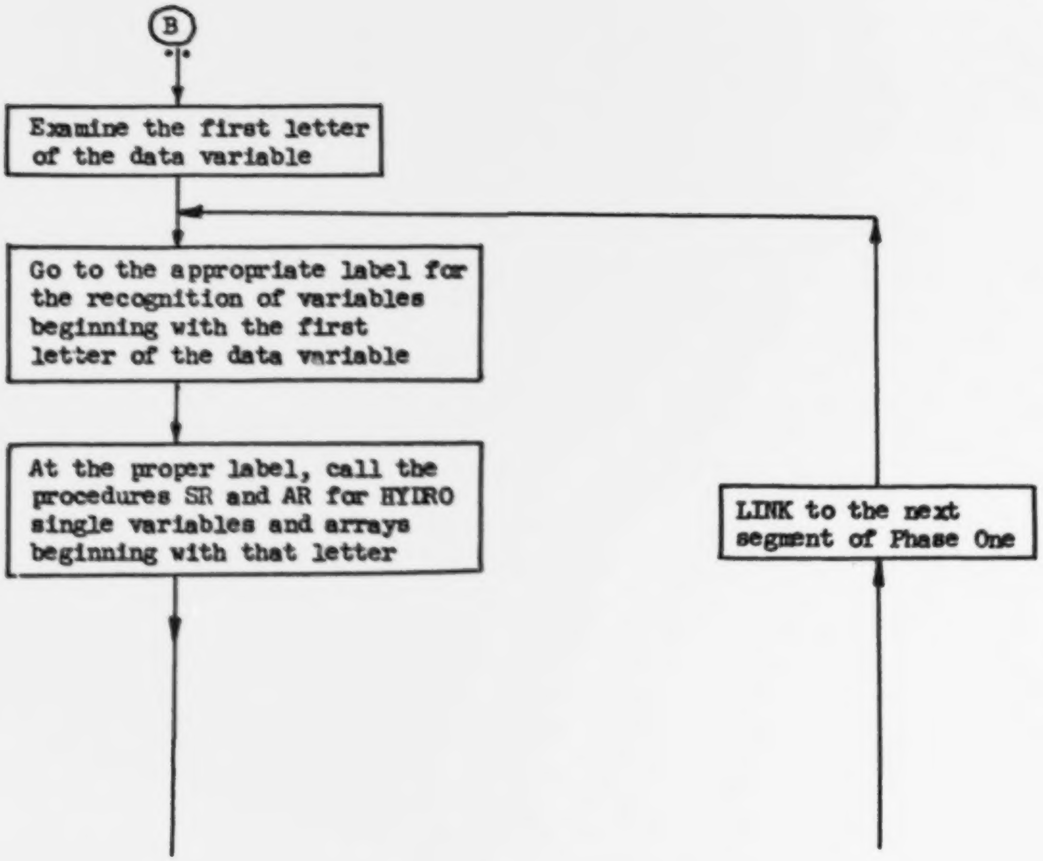
3.1 Flow Chart


Enter HYDRO program to computer
First compiler segment automatically loaded into computer core along with the ALGOL translator to run as an ALGOL program
PHASE ONE
Declare variables used by Phase 1
Declare the procedures NEXT.COLUMN, PR, SR, AR used in Phase 1
Initialize variables
Read a card into IMAGE
A
Locate the beginning and the end of a command or variable in IMAGE

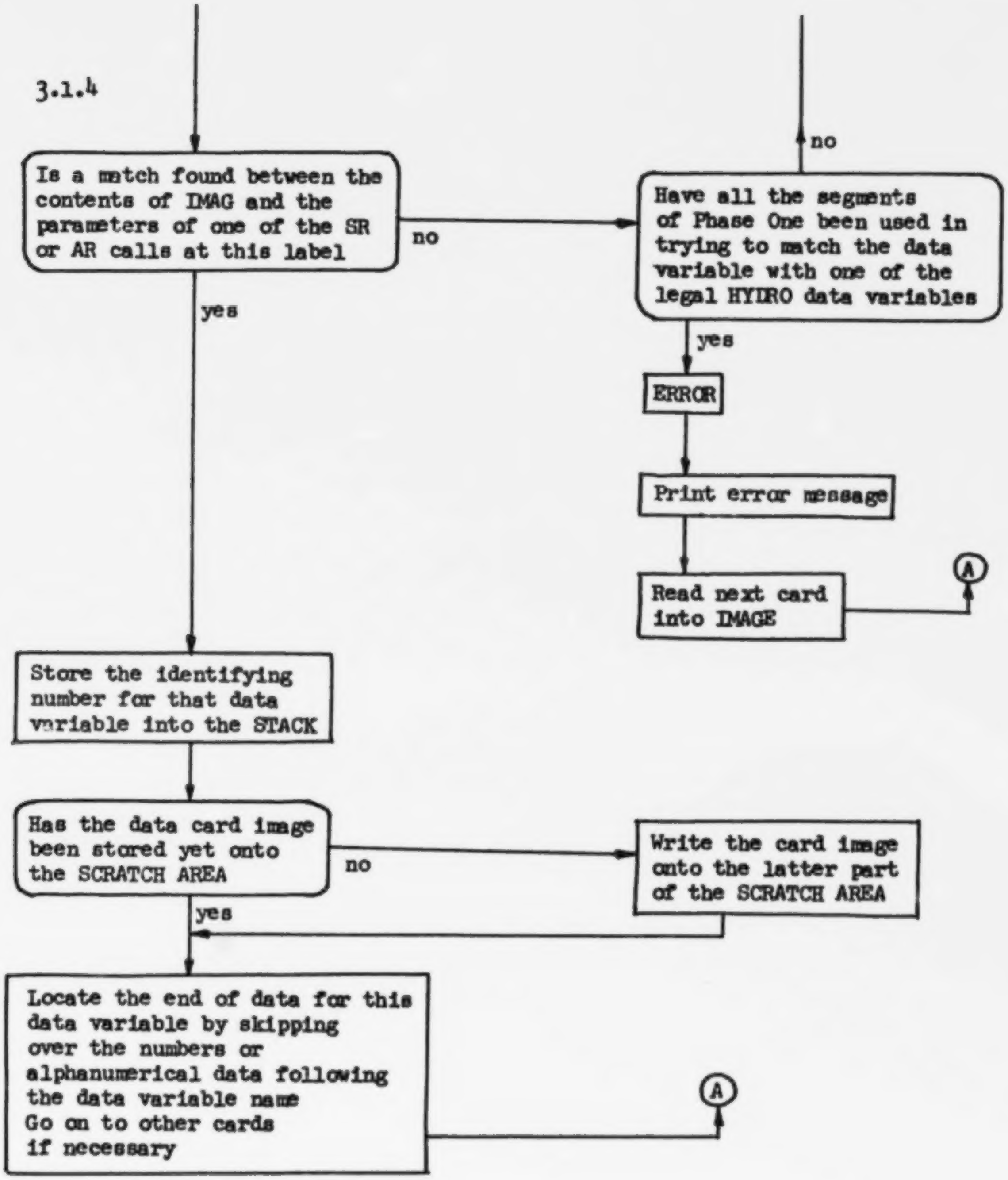
3.1.2


Store that command or variable into IMAG
Examine the remainder of the card IMAGE in search of a left arrow indicating a data variable
Does this card contain a data variable
yes
B
no
Then it contains a procedure call. Examine the first letter of the procedure call
Go to the appropriate label for the recognition of procedures beginning with the first letter of the procedure call
At the proper label, call the procedure PR for the HYDRO procedure calls beginning with that letter
Is a match found between the contents of IMAG and the parameters of one of the PR calls at this label
no
yes
LINK to the next segment of Phase One
no
Have all the segments of Phase One been used in trying to match the procedure call with one of the procedures in the HYDRO library
yes
ERROR No match found
A
Print error message
Read next card into IMAGE

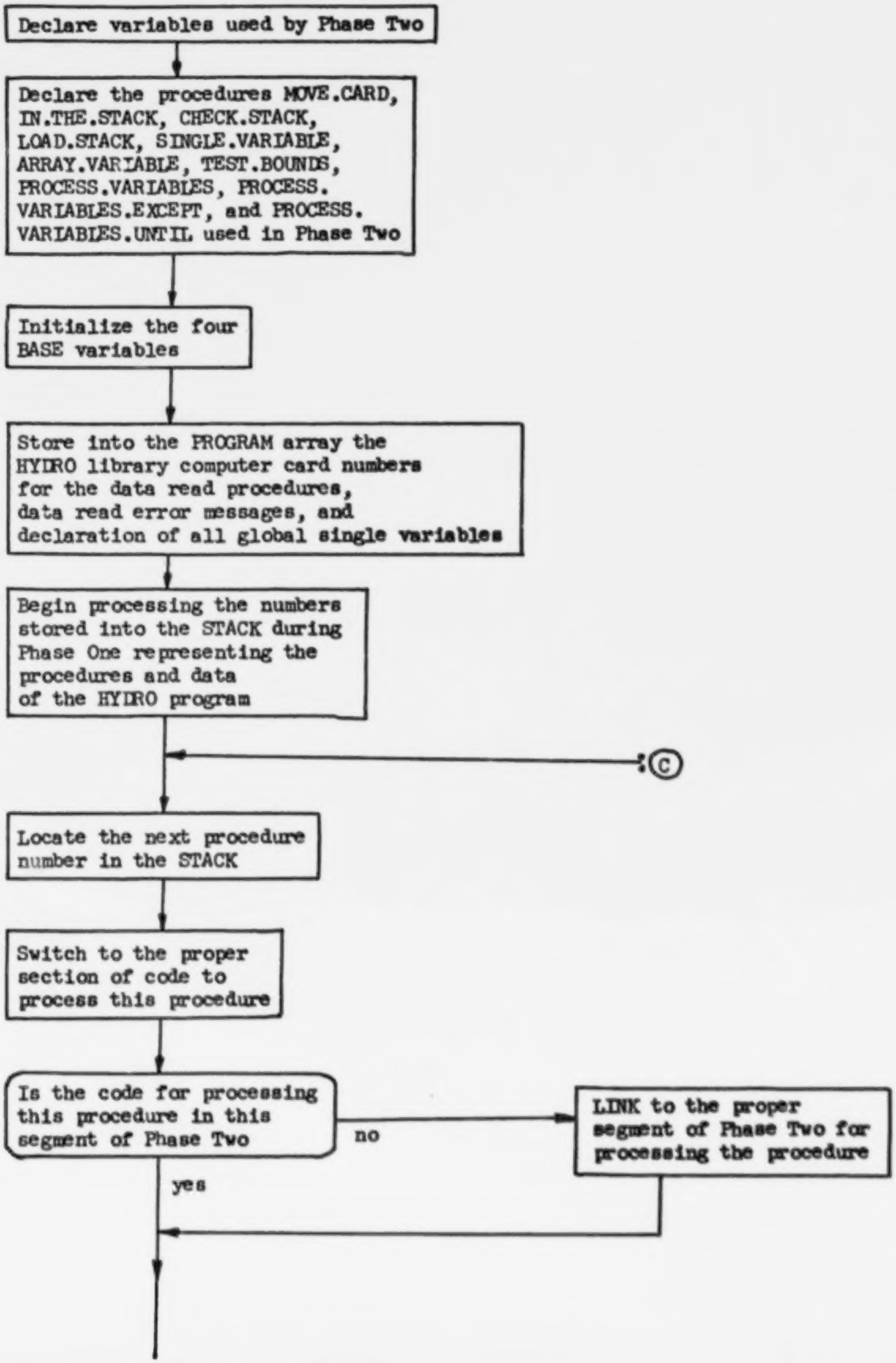

Store the identifying number for that procedure into the STACK
Is the command PROGRAM.END
no
Read next card into Image
A
yes
Were there any ERRORS detected during Phase One
yes
HALT
no
LINK to the first Segment of Phase Two


B
Examine the first letter of the data variable
Go to the appropriate label for the recognition of variables beginning with the first letter of the data variable
At the proper label, call the procedures SR and AR for HYDRO single variables and arrays beginning with that letter
LINK to the next segment of Phase One

3.1.4


Is a match found between the contents of IMAG and the parameters of one of the SR or AR calls at this label
no
yes
Have all the segments of Phase One been used in trying to match the data variable with one of the legal HYDRO data variables
no
yes
ERROR
Print error message
Read next card into IMAGE
A
Store the identifying number for that data variable into the STACK
Has the data card image been stored yet onto the SCRATCH AREA
no
yes
Write the card image onto the latter part of the SCRATCH AREA
Locate the end of data for this data variable by skipping over the numbers or alphanumerical data following the data variable name Go on to other cards if necessary
A

PHASE TWO



Declare variables used by Phase Two

Declare the procedures MOVE.CARD, IN.THE.STACK, CHECK.STACK, LOAD.STACK, SINGLE.VARIABLE, ARRAY.VARIABLE, TEST.BOUNDS, PROCESS.VARIABLES, PROCESS.VARIABLES.EXCEPT, and PROCESS.VARIABLES.UNTIL used in Phase Two

Initialize the four BASE variables

Store into the PROGRAM array the HYDRO library computer card numbers for the data read procedures, data read error messages, and declaration of all global single variables

Begin processing the numbers stored into the STACK during Phase One representing the procedures and data of the HYDRO program

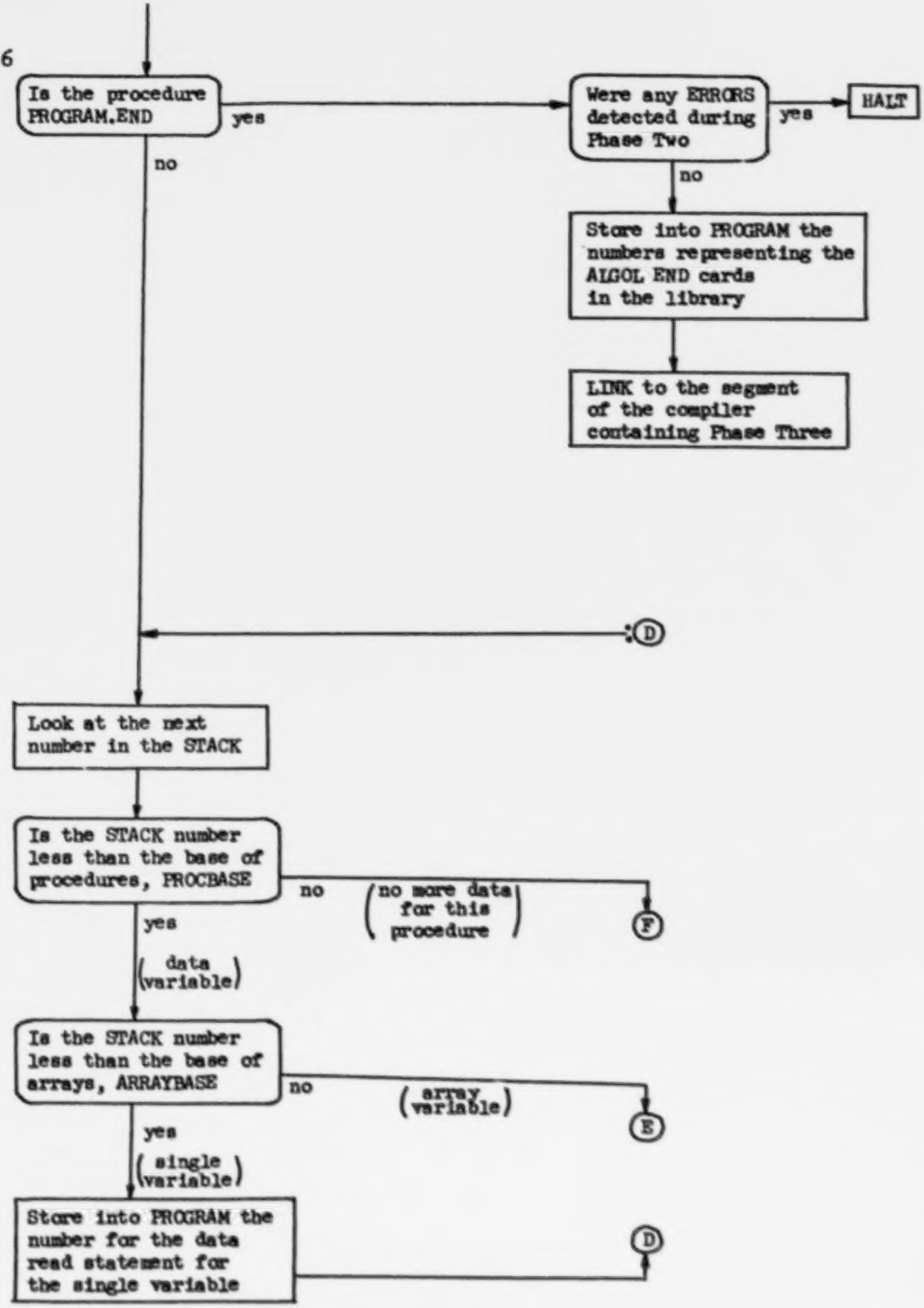
Locate the next procedure number in the STACK

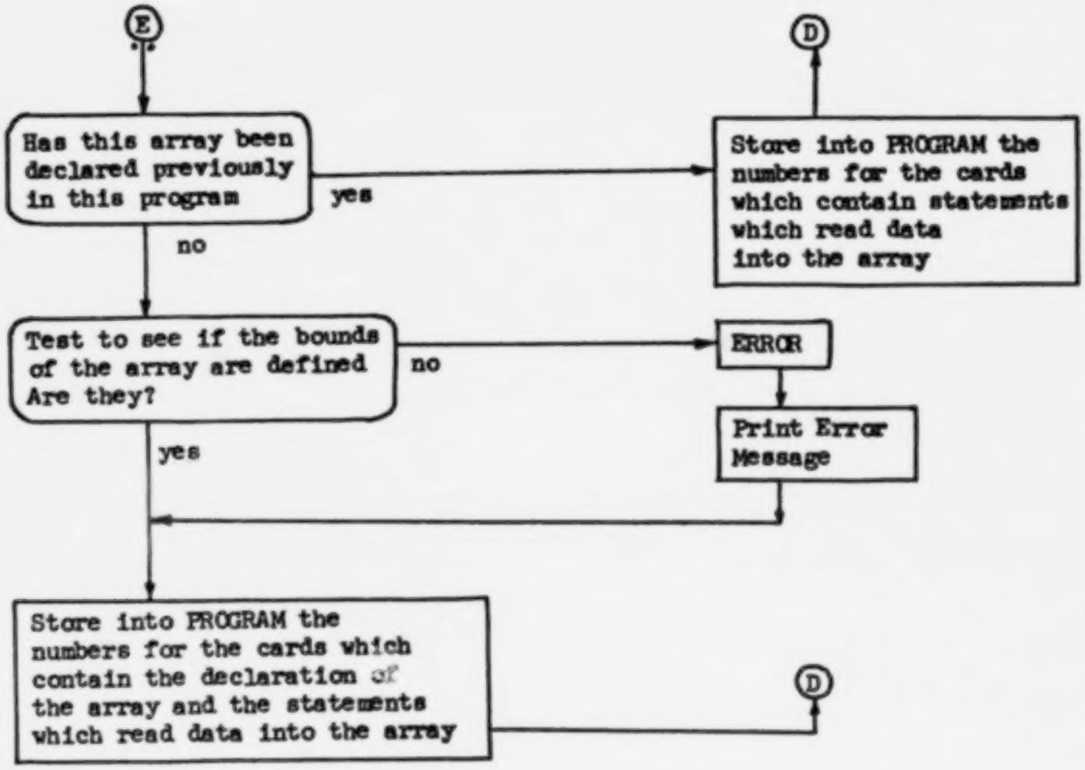
Switch to the proper section of code to process this procedure

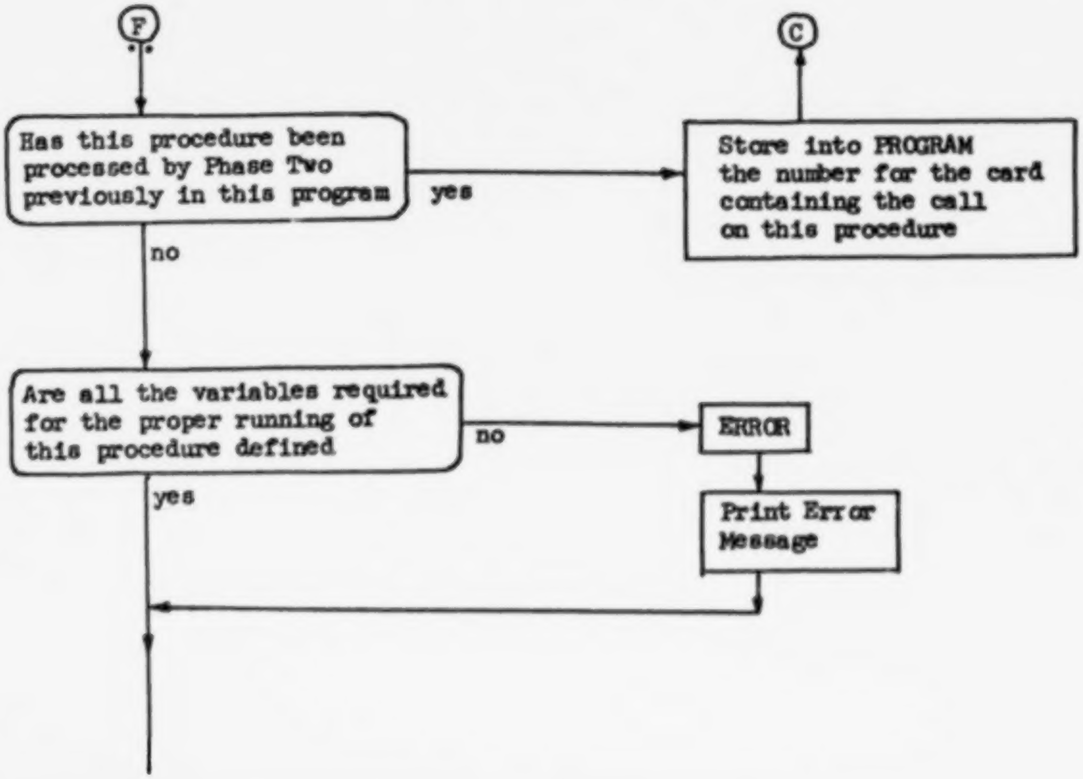
Is the code for processing this procedure in this segment of Phase Two

LINK to the proper segment of Phase Two for processing the procedure

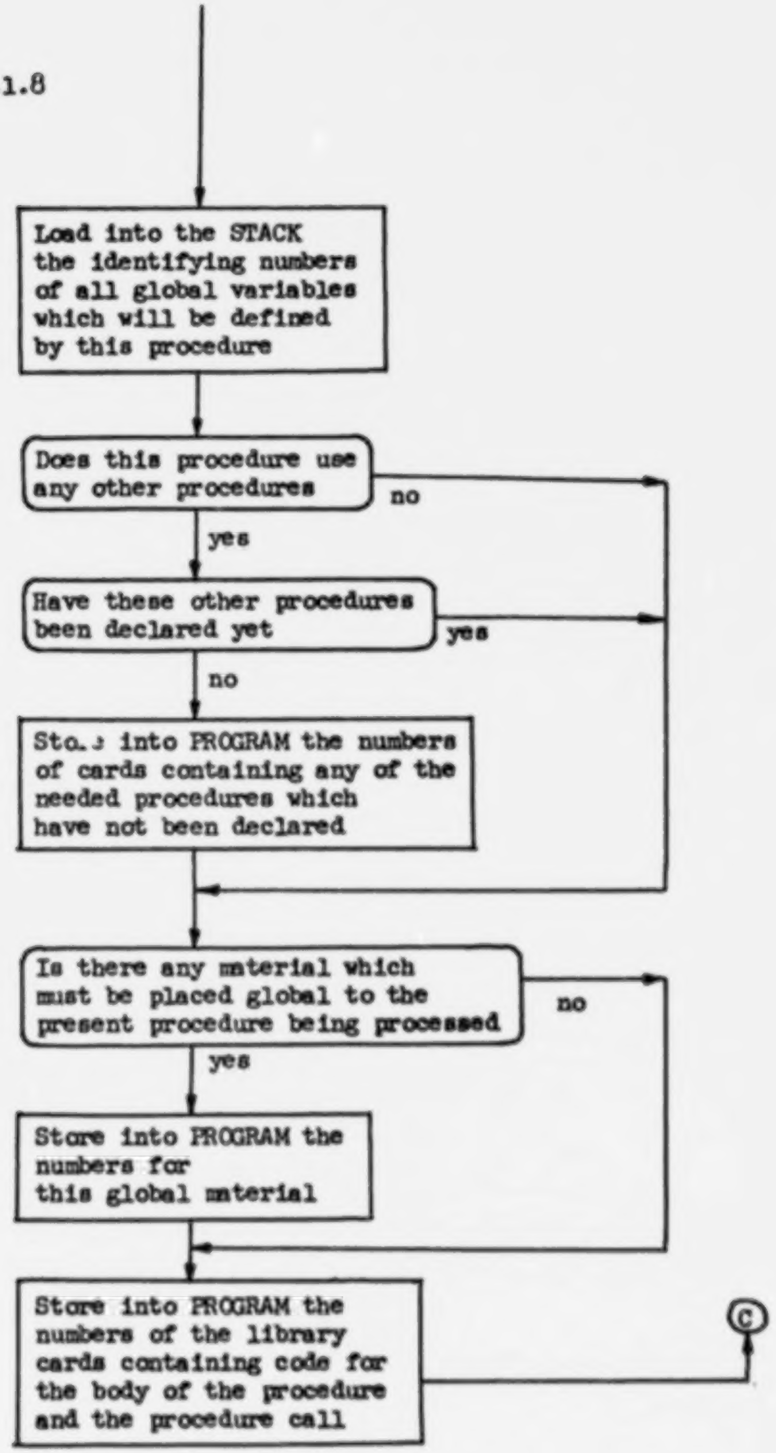
3.1.6

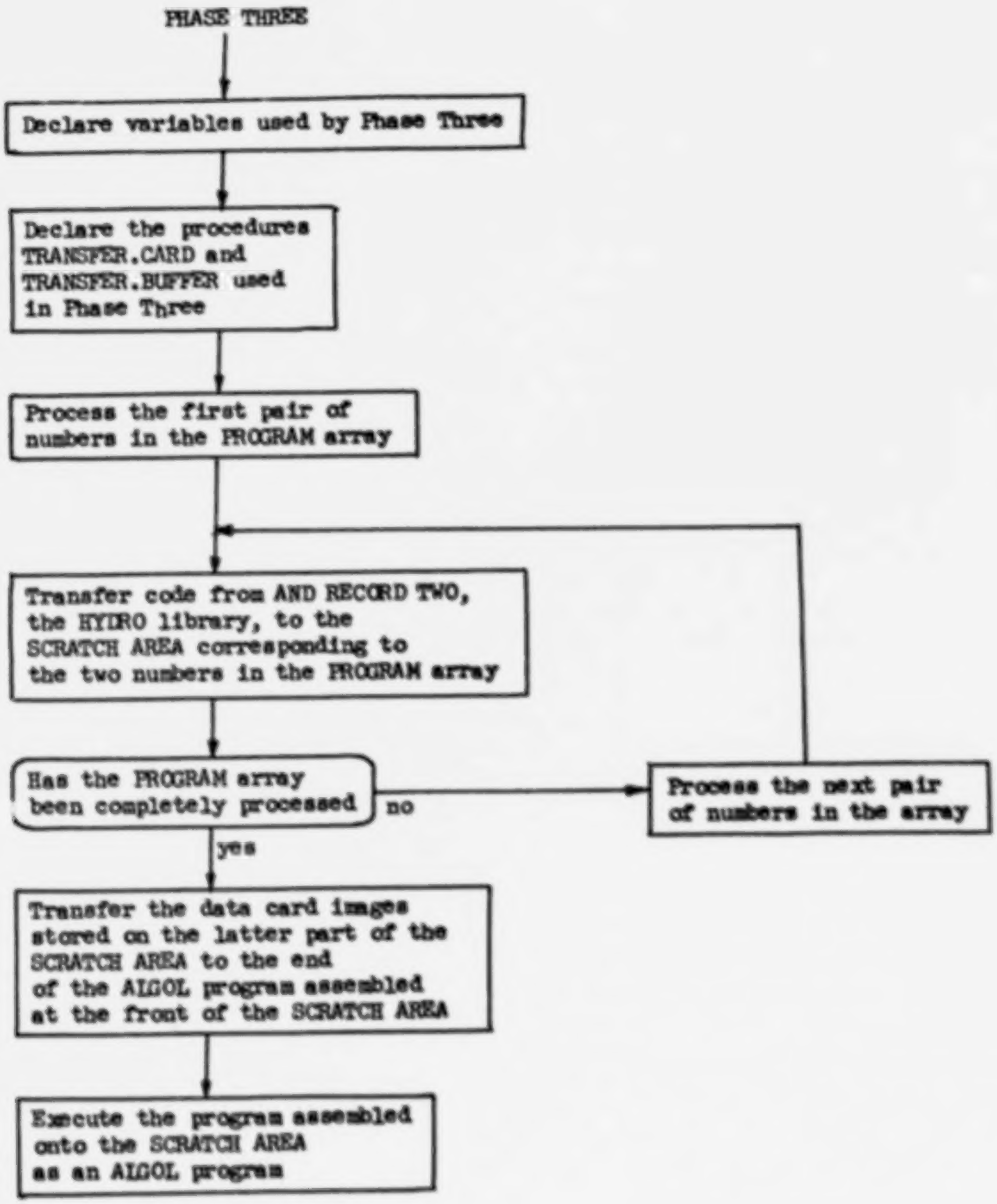

Is the procedure PROGRAM.END
yes
no
Were any ERRORS detected during Phase Two
yes
HALT
no
Store into PROGRAM the numbers representing the ALGOL END cards in the library
LINK to the segment of the compiler containing Phase Three
D
Look at the next number in the STACK
Is the STACK number less than the base of procedures, PROCBASE
no
(no more data for this procedure)
F
yes
(data variable)
Is the STACK number less than the base of arrays, ARRAYBASE
no
(array variable)
E
yes
(single variable)
Store into PROGRAM the number for the data read statement for the single variable
D

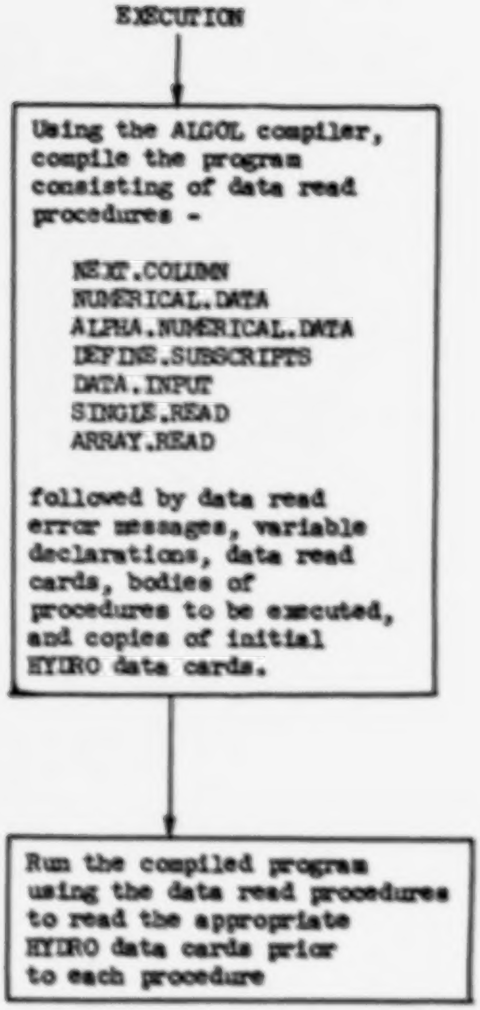

E
Has this array been declared previously in this program
yes
Store into PROGRAM the numbers for the cards which contain statements which read data into the array
D
no
Test to see if the bounds of the array are defined Are they?
no
ERROR
Print Error Message
yes
Store into PROGRAM the numbers for the cards which contain the declaration of the array and the statements which read data into the array
D


F
Has this procedure been processed by Phase Two previously in this program
yes
Store into PROGRAM the number for the card containing the call on this procedure
C
no
Are all the variables required for the proper running of this procedure defined
no
ERROR
Print Error Message
yes

3.1.8


Load into the STACK the identifying numbers of all global variables which will be defined by this procedure
Does this procedure use any other procedures
no
yes
Have these other procedures been declared yet
yes
no
Store into PROGRAM the numbers of cards containing any of the needed procedures which have not been declared
Is there any material which must be placed global to the present procedure being processed
no
yes
Store into PROGRAM the numbers for this global material
Store into PROGRAM the numbers of the library cards containing code for the body of the procedure and the procedure call
C


PHASE THREE
Declare variables used by Phase Three
Declare the procedures TRANSFER.CARD and TRANSFER.BUFFER used in Phase Three
Process the first pair of numbers in the PROGRAM array
Transfer code from AND RECORD TWO, the HYDRO library, to the SCRATCH AREA corresponding to the two numbers in the PROGRAM array
Has the PROGRAM array been completely processed
no
Process the next pair of numbers in the array
yes
Transfer the data card images stored on the latter part of the SCRATCH AREA to the end of the ALGOL program assembled at the front of the SCRATCH AREA
Execute the program assembled onto the SCRATCH AREA as an ALGOL program



EXECUTION

Using the ALGOL compiler, compile the program consisting of data read procedures -

NEXT.COLUMN
NUMERICAL.DATA
ALPHA.NUMERICAL.DATA
DEFINE.SUBSCRIPTS
DATA.INPUT
SINGLE.READ
ARRAY.READ

followed by data read error messages, variable declarations, data read cards, bodies of procedures to be executed, and copies of initial HYDRO data cards.

Run the compiled program using the data read procedures to read the appropriate HYDRO data cards prior to each procedure

3.2 Explanation of the Flow Chart

The placement of the word HYDRO in the language field of the input deck job card causes the compiler to be entered automatically by a LINK to the first segment of Phase One. Thus the PHASE.ONE.RECOGNIZE procedure is placed into core along with the ALGOL translator and is run as an ALGOL program. The HYDRO input deck, entered from cards or teletype, serves as the data for this program.

Phase One begins by declaring variables global to the entire compiler, and then it declares four other procedures used by itself - NEXT.COLUMN, PR, SR, and AR. The first procedure is a card scanner, and the latter three perform "procedure recognition," "single variable recognition," and "array variable recognition."

Computation begins in Phase One with the initialization of important variables, and then NEXT.COLUMN is called for the first time in order to read a HYDRO input card into the IMAGE vector. As Phase One proceeds, each of the input cards is read into this vector as an alphanumeric string, scanned, and the contents recognized by PR, SR, or AR. A record of what is recognized is built up as a string of identifying numbers in a list called the STACK. This list is later used by Phase Two to assemble an ALGOL program which is the equivalent of the HYDRO input code.

Each time a new card is read into IMAGE, code is executed in order to locate on the card the columns at which a procedure call or a variable begin and end. This action is performed at label A of the flow chart. The procedure scans the card until the first character is located. It then proceeds further until a blank, a left arrow, or a left bracket is found. Once the procedure call or variable is bounded, the characters which comprise it are transferred to another logic vector, IMAG, whose name symbolizes the fact that it is part of the IMAGE vector. Then the characters of IMAGE immediately following what was stored into IMAG are examined. If a left bracket is found in one of these indicating a subscripted variable, code is executed to move past the contents of the brackets to find a left arrow. If a left arrow is finally found, control passes to the part of Phase One where data variables are recognized. This is symbolized by the label B on the flow chart. If the end of the card is reached without finding a left arrow, the process of procedure recognition begins.

The first letter of a procedure call in IMAG is examined to see which letter of the alphabet it is; for each letter of the alphabet there is a label in PHASE.ONE.RECOGNIZE for procedure recognition. Control then passes to the appropriate label. At the label for each letter are calls on the procedure PR for some or all of the legal HYDRO procedures beginning with that letter. (Some may be located in other segments of Phase One.) These calls are made one by one until a match is found between the characters in IMAG and the characters which are the parameters of a particular call on PR. If no match is found after all calls are made on PR at a given label, LINKS are made to the other Phase One segments to check if the inputted procedure name can be recognized there. If all Phase One segments are checked, and no match is found, an error message is given, a new card is read into IMAGE, and control returns to A. If a match is found in one of the segments, the base of procedures, PROCBASE, plus the identifying number of the procedure are placed into the current position of the STACK.

If a legitimate command is recognized, special action must be taken if it is the PROGRAM.END command. This signals that the end of the HYDRO program has been reached, and a LINK can be made to Phase Two if no errors were found in Phase One. If errors were found, the compiler halts at this point. If the procedure recognized was not PROGRAM.END, the next HYDRO card is read into IMAGE, and control returns to label A of the flow chart.

The code for recognizing data variables, very similar to that for recognizing procedures, is located at B. The first letter of the data variable is examined, and control passes to the appropriate label for recognizing data variables beginning with that letter. At the proper label, calls are made on SR and AR for legal HYDRO data variables beginning with that letter. If no match is found, LINKs are made to other Phase One segments and the search is continued. If all segments are finally checked, and no match is found at the proper label, it means the input variable was not a legal HYDRO one, and an error message is given. A new card is read into IMAGE and control passes up to A. On the other hand, if a match is found, the identifying number for the data variable is placed into the current position of the STACK. Before returning to A, a check is made if the image of the data card being scanned has previously been written onto the end of the SCRATCH AREA. All data cards are placed there since they will again be used as data for the execution of the ALGOL program assembled from the HYDRO inputs. If the card image has not been previously placed there, if is written there at this time. Then code is executed to move the pointer on the IMAGE vector past all the actual data following the left arrow on the card. To do this, a search is made for the semicolon which signals the completion of data input for any data variable. More cards may have to be read into IMAGE in order to find this, if the data for the variable lie on more than one card. When the semicolon is finally located, control returns to A without reading a new card into IMAGE. This action is omitted because more than one data variable may appear on a card.

Phase One continues functioning, recognizing procedure calls, data variables, and finding errors, until finally the PROGRAM.END card is found. If no errors have been located by the time this card is reached, a LINK is made to the first segment of Phase Two.

The procedure comprising the body of Phase Two is called PHASE.TWO. ASSEMBLE. It replaces PHASE.ONE.RECOGNIZE in core and runs as an ALGOL program. At the beginning of this procedure, all of the sub-procedures used in Phase Two are declared: MOVE.CARD, IN.THE.STACK, CHECK.STACK, LOAD.STACK, SINGLE.VARIABLE, ARRAY.VARIABLE, TEST.BOUNDS, PROCESS. VARIABLES, PROCESS.VARIABLES.EXCEPT, and PROCESS.VARIABLES.UNTIL.

Once these procedures are declared, the fundamental job of Phase Two can begin; that of building up in the array PROGRAM a list of number pairs which represent the blocks of code on the language library record, i.e. AND RECORD TWO, which, when placed back to back, will form a legal ALGOL program equivalent to that in HYDRO entered by the user. These pairs of card image numbers are always stored into the PROGRAM array by a call on the procedure MOVE.CARD.

Execution of Phase Two begins with the initialization of the four BASE variables, BASE1, BASE2, BASE3, and BASE4, representing the division points between important sections of the language library. The first number pair is then stored into the PROGRAM array; this pair represents the card images bounding the data read procedures, the data read error messages, and the declarations of all global single variables. This code is automatically placed first in every ALGOL program assembled from HYDRO. Then, at the label C, the processing of the numbers in the STACK which represent the HYDRO program begins. Each procedure number located there and the data numbers following it are processed in somewhat the same way. In brief, control passes to the processing code for the procedure, the variable numbers following it are all processed, the STACK is checked to see if everything necessary is defined, the STACK is loaded with the new variables which will be defined, and finally the numbers are stored into PROGRAM for the code to execute the procedure.

In more detail, a procedure is processed as follows: First, a switch is made to the proper section of processing code for the procedure. Because there are several segments which comprise Phase Two, a LINK may have to be made at this point to one of the other Phase Two segments in order to locate the proper section of processing code. Having done this where necessary, a check is made to see if the procedure number being processed represents PROGRAM.END. If it does, a check is made to see if any errors were found in Phase Two. If there were any, the compiler halts at this point. If there were none, all unmatched BEGINs in the code represented by the numbers in the PROGRAM array are matched with ENDs by storing at the end of PROGRAM the card numbers representing the six cards of ALGOL ENDs located at the beginning of the language library. Control then passes to Phase Three. If a procedure number does not represent PROGRAM.END, processing begins at label D.

At D, the variable numbers following the procedure number in the STACK are first processed (by PROCESS.VARIABLES). A check is made for each variable number to see if it is less than the PROCBASE. If it is not, then another procedure number has been located, and processing of variables is completed for the initial procedure. Control then passes to F for the processing of the procedure proper.

On the other hand, if the number in the STACK is less than the PROCBASE, then it represents a data variable, either a single variable or an array. A check is then made to see if it is less than the ARRAYBASE. If it is not, it represents an array, and control passes to E on the flow chart, i.e. the ARRAY.VARIABLE procedure is called. However, if the STACK number is less than the ARRAYBASE, then SINGLE.VARIABLE is called. In the procedure SINGLE.VARIABLE, the card numbers for code which will read data into the single variable represented by the number in the STACK are stored into PROGRAM by a call on MOVE.CARD. When this has been done, the next number in the STACK is processed, at D on the flow chart.

At E is located the flow chart for the ARRAY.VARIABLE procedure, executed when an array variable number is found in the STACK. A check is first made if the array has ever been declared previously in the ALGOL program being assembled. If it has, the variable DECLARED is set TRUE, and because of this the number pair is stored into PROGRAM representing code which will only read data into the array. Control then returns to D to process the next number in the STACK. If, on the other hand, the array has not been previously declared, the variable DECLARED is made FALSE. This causes first a call on the procedure TEST.BOUNDS to check if all of the bounds of the array have been defined. If they have not, an error message is given to the user. After TEST.BOUNDS is called, MOVE.CARD is called to store into PROGRAM the number pair representing code which will both declare the array and read data into it.

Then control passes back to D to process the following number in the STACK.

Data numbers are continually processed by PROCESS.VARIABLES, either as single variables or as array variables, until another procedure number is found in the STACK. This signals the end of the data for the initial procedure and tells the compiler that the remainder of the processing for that procedure can be carried out.

At F is the flow chart for the compiler code executed in processing a procedure number once all the variables for that procedure have been processed. First, a check is made to see if this procedure has ever been processed before. If it has, the procedure body and all necessary code global to it have already been assembled into the ALGOL program represented by the library card numbers in PROGRAM. Therefore all that is required is a call on the procedure. The card number for the procedure call is placed into PROGRAM, and control passes back to C to process the next procedure number in the STACK.

If, on the other hand, the procedure has not been previously processed, checks are made to see if all variables required for the proper running of the procedure are defined. This is accomplished by a succession of calls on the procedure CHECK.STACK, since the STACK is also the defined list for variables. If any variables which are necessary are found not to be defined, an error message is printed. After checks have been made on all required variables, the identifying numbers of all global variables which this procedure will define (and which may be used in later procedures) are loaded into the STACK. The LOAD.STACK procedure is used for this.

After the STACK has been checked and loaded, the numbers for cards in the library containing code necessary to run the procedure can be placed into PROGRAM. First, if other procedures are used by the procedure being processed, the numbers representing their code must be placed into PROGRAM. Then the numbers for any piece of code which must be global to the procedure being processed are put into PROGRAM. Finally, the numbers representing the body of the procedure and its call are assembled into PROGRAM. This completes the processing of the procedure, and the compiler goes back to C in the flow chart to process the following procedure number in the STACK.

This process of recycling back to C continues as long as procedure numbers occur in the STACK. When finally a procedure number in the STACK corresponds to the PROGRAM.END call, a LINK is made to Phase Three, if no errors have been detected.

The procedure comprising the body of Phase Three is called PHASE. THREE.TRANSFER. It replaces Phase Two, and begins by declaring TRANSFER. CARD, its principal procedure. Within TRANSFER.CARD, a smaller procedure, TRANSFER.BUFFER, is declared. The procedure TRANSFER.CARD is then called for each pair of numbers in the PROGRAM array. This transfers from the language library to the SCRATCH AREA the code represented by the PROGRAM number pairs. When the entire PROGRAM array has been processed in this way, that is, the complete ALGOL program is assembled on the SCRATCH AREA, the HYDRO input data in temporary storage at the end of the SCRATCH AREA is transferred forward to the end of the ALGOL program. This completes the transferral of code onto the SCRATCH AREA; the program there is ready to run. A short program in the WHAT language, explained in Section 9 of Appendix I, is used at the end of Phase Three to execute the program from the SCRATCH AREA.

During execution the seven data reading procedures, assembled at the head of the ALGOL program, read data from the original HYDRO data cards placed at the end of the program. These reads are accomplished by calls on SINGLE.READ and ARRAY.READ placed throughout the ALGOL program for the various variables given by the user as input.

4. COMPILER PROCEDURES AND VARIABLES

This Chapter identifies all of the procedures and variables used in the HYDRO compiler. The first section of the Chapter deals with procedures, and the latter with variables. In each of the sub-sections of the Chapter, a list is first given of the procedures or variables explained there, followed by the actual explanations or definitions. These can be used while studying the compiler whenever any doubt arises concerning the purpose or function of a particular procedure or variable.

4.1 Procedures

Computer Center Library Procedures

LINK
ANDFILE
DISCREAD
DISCWRITE

In the explanations of the procedures which follow, it is assumed that the reader is somewhat familiar with the "AND" record storage system of the Carnegie Tech Computation Center. This system is explained more fully in several sections of Appendix I.

LINK - The LINK procedure is used for the purpose of moving one of the segments of the compiler from storage on tape into the computer core. Such a segment would replace the segment already in core and then run as an ALGOL program. The number parameter of the LINK procedure call specifies which segment is to be transferred.

ANDFILE - The ANDFILE procedure establishes a certain AND RECORD of a particular user as a numerical file to be referred to in calls on DISCREAD and DISCWRITE. Each call on ANDFILE must be given five parameters. The first gives the user number, i.e. the number designating the person whose records are to be accessed or written on. The second parameter tells which program on these records is desired. The logical file number to be associated with this AND file is given by the third parameter. The first element of an available array of at least 320 words is the fourth parameter, and the last parameter is an error exit for the procedure to switch to in case trouble is encountered.

DISCREAD - With the DISCREAD procedure, code can be read from a logical file, i.e. an AND RECORD on either tape or disc, into an ALGOL array in the computer core. DISCREAD also has five parameters. The first tells how many four-character alphanumeric words are to be read. The array element into which the first word is to be read is the next parameter given. The third tells what logical file, or AND RECORD, the material is to be read from. The next is the block (of 320 words) after which reading is to begin, and the last parameter is an error exit similar to that in ANDFILE.

DISCWRITE - The DISCWRITE procedure enables code to be transferred or written from an ALGOL array in the computer core onto an AND RECORD or the AND SCRATCH AREA. Five parameters, very similar to those used with DISCREAD, are required for each DISCWRITE call. They consist of the number of words to write, the ALGOL array from which code will be written, the logical file number of the file to be written onto, the block after which writing should commence, and finally the error exit.

Phase One Procedures

NEXT.COLUMN
PR(N,D1,D2,D3,---,D30)
SR(N,D1,D2,D3,---,D15)
AR(N,D1,D2,D3,---,D15)

NEXT.COLUMN - NEXT.COLUMN is the card scanning procedure. Each time it is called it moves the pointer, I, on the alphanumeric vector, IMAGE, one column to the right so that the contents of that column can be examined. When it finds that the pointer has reached column 81 of the card image, it reads logically the following HYDRO input card into IMAGE and resets the pointer to one.

PR(N,D1,D2,D3,---,D30) - PR is the procedure which recognizes HYDRO procedure calls. It has 31 parameters: a number and 30 characters representing one of the legal procedure calls of HYDRO. In a call on PR, the 30 parameter characters are compared with the contents of the IMAG vector, which has in it the characters of the HYDRO procedure call submitted by the user. If all the characters in IMAG match the parameters of PR, the variable PROCBASE plus the number representing the procedure call is placed into the STACK.

SR(N,D1,D2,D3,---,D15) - Similar to PR, SR is the procedure which recognizes single read-in variables from the HYDRO input code. When a data variable is scanned in the code it is also stored into the IMAG vector. Calls on SR are then made to attempt to match the contents of IMAG with one of the legal HYDRO single variables. SR has only 16 parameters: a number and 15 characters representing one of the legal single variables. When a match is found by a call on SR, the number parameter is stored into the STACK.

AR(N,D1,D2,D3,---,D15) - The AR procedure is almost identical to SR. It has the same number of parameters and works in a similar way. The function of AR is the recognition of read-in arrays. When it recognizes an array, the number parameter plus the value of the variable ARRAYBASE is placed into the STACK.

Phase Two Procedures

MOVE.CARD (A,B)
IN.THE.STACK (A)
CHECK.STACK (A)
LOAD.STACK (A)
SINGLE.VARIABLE
ARRAY.VARIABLE
TEST.BOUNDS (A,B,C)
PROCESS.VARIABLES
PROCESS.VARIABLES.EXCEPT (A)
PROCESS.VARIABLES.UNTIL (A)

MOVE.CARD (A,B) - The purpose of the MOVE.CARD procedure is to store into the PROGRAM array a pair of numbers representing a section of code to be moved from the language library to the SCRATCH AREA. The two numbers are the parameters of the MOVE.CARD call. Each time Phase Two establishes that certain code is required from the library it calls MOVE.CARD of the card numbers which bound that code. MOVE.CARD first increments the bound on PROGRAM, PAIRS, by one. It then stores the two parameters of the MOVE.CARD call into the next two locations of the PROGRAM array. The actual transferral of the code represented by the numbers in PROGRAM does not take place until Phase Three.

IN.THE.STACK (A) - IN.THE.STACK is a Boolean procedure which checks to see if a number given as the parameter to the procedure is located in the STACK. Procedure numbers found in the STACK represent procedures that have been declared. Variable numbers in the STACK mean the variable is both declared and defined. This procedure yields the value TRUE if the number is in the STACK, and FALSE if it is not.

CHECK.STACK (A) - CHECK.STACK is merely an extension of IN.THE.STACK. It calls IN.THE.STACK for its parameter, and if a value of FALSE is given, it outputs an error message to the user.

LOAD.STACK (A) - The purpose of LOAD.STACK is to place into the STACK the identifying number for a global variable which will be declared and defined by the procedure being processed. This number is given as the parameter to the procedure.

SINGLE.VARIABLE - SINGLE.VARIABLE is a procedure called within the PROCESS.VARIABLES procedure. It is called when a number is encountered in the STACK which represents a single variable from one of the data cards. It calls MOVE.CARD for the card image in the language library which will read data into that single variable.

ARRAY.VARIABLE - The ARRAY.VARIABLE procedure is called by the PROCESS. VARIABLES procedures, and performs somewhat the same function as SINGLE.VARIABLE, but for an array number in the STACK. The procedure uses a switch, ARRY, and another procedure, TEST.BOUNDS. First, a check is made to see if the declaration of the array has been assembled previously in the program. If it has, the variable DECLARED is set TRUE; if not, FALSE. If the array has not been declared, a switch is made to the appropriate ARRY label, and a call is made on the procedure TEST.BOUNDS to see if the bounds of the array are defined. MOVE.CARD is then called for the code which will declare the array and read data into it. If the array has been declared, then MOVE.CARD is simply called for the code to read data into the array.

TEST.BOUNDS (A,B,C) - TEST.BOUNDS is a procedure declared and used only within the ARRAY.VARIABLE procedure. Its purpose is to check by means of several calls on the procedure IN.THE.STACK to see if the bounds of an array are defined. The numbers representing the bounds of the array are given as the parameters to this procedure. If one or more of the bounds is not defined, an error message is given to the user.

PROCESS.VARIABLES - The PROCESS.VARIABLES procedure is usually the first called within any of the sections of Phase Two code used for processing the different HYDRO procedures. Its purpose is to examine all of the data variable numbers in the STACK following a particular procedure number, and by calling SINGLE.VARIABLE or ARRAY.VARIABLE, execute MOVE.CARD for the proper code which will declare and read data into those variables global to the procedure body. PROCESS.VARIABLES handles all variable numbers in the STACK until it finds another procedure number.

PROCESS.VARIABLES.EXCEPT (A) - This procedure is sometimes called in place of PROCESS.VARIABLES. It functions in exactly the same way as PROCESS.VARIABLES except that it will not process the variable number given as its parameter. When it reaches such a number in the STACK it moves the STACK pointer, OPNO, past the remaining variable numbers to the position in the STACK where the next procedure number is located. The variable numbers bypassed are not processed because it is desired that the read statements for them be placed within the procedure body rather than global to it. At the proper time in the processing code for the procedure, MOVE.CARD is executed for these read statements.

PROCESS.VARIABLES.UNTIL (A) - This procedure also replaces PROCESS. VARIABLES on certain occasions. It works in the same way as PROCESS. VARIABLES until it reaches the variable number in the STACK given by its parameter. It then stops without moving the STACK pointer ahead to the next procedure number. This is done because PROCESS. VARIABLES.UNTIL will always be followed by a call on PROCESS.VARIABLES before the end of the processing code for the procedure. This combination of PROCESS.VARIABLES.UNTIL and PROCESS.VARIABLES is used when some other code has to be assembled from the library within the data reading code. When it is necessary to use this combination, variables are processed until the point is reached when the material within the data reading code must be assembled. That code is then assembled, and finally the processing of variables is completed up to the next procedure number in the STACK.

Phase Three Procedures

TRANSFER.BUFFER
TRANSFER.CARD (A,B)

TRANSFER.BUFFER - TRANSFER.BUFFER is a small procedure used internally within the principal procedure of Phase Three, TRANSFER.CARD. Its purpose is to transfer a desired portion of the contents of an array READBUFFER into another array WRITEBUFFER, and to DISCWRITE the WRITEBUFFER onto the SCRATCH AREA if the contents of WRITEBUFFER equal 1600 words.

4.1.8

TRANSFER.CARD (A,B) - The procedure TRANSFER.CARD operates with a single pair of numbers from the PROGRAM array each time it is called. Its function is to move the code represented by that pair of numbers from the language library, AND RECORD TWO, to the AND SCRATCH AREA. TRANSFER.CARD computes the lower and upper bounds of the material to be transferred in terms of logic words. It then loads successive sections of the library into the READBUFFER by using the DISCREAD routine. Appropriate parts of the READBUFFER are then transferred to the WRITEBUFFER by the procedure TRANSFER.BUFFER. When the WRITEBUFFER is filled it is written onto the SCRATCH AREA by the use of DISCWRITE. Code is moved continually in this way until all the code bounded by the parameters to the procedure is transferred.

4.2 Variables

Variables for the Entire Compiler

I, J, K
STACK [1:1000]
PROGRAM [1:500,1:2]
PAIRS
ARRAYBASE
PROCBASE
SPECBASE
DATABLOCK
ERROR

The above variables are all used in more than one of the compiler Phases.

I, J, K - These are iteration counters used throughout the compiler. In Phase One they are used to mark the progress of a pointer moving across the input card IMAGE in order to locate the beginning and the end of the procedure call or variable name located there.

STACK [1:1000] - The STACK is a vector containing a record in numbers of the procedures called, the variables defined by read-ins, and the variables defined by procedures. Above the 500th element of the STACK, Phase One stores numbers identifying the procedures called and data variables read into by the input program. Below the 500th element, Phase Two stores numbers representing global variables defined by the procedures. A scan of the STACK at any time during the assembly of the ALGOL program will tell what procedures have been declared and what variables have been defined up to that time in the assembly of the program.

PROGRAM [1:500, 1:2] - The PROGRAM array contains the library card numbers which represent the sections of ALGOL code to be transferred from the language library, AND RECORD TWO, to the AND SCRATCH AREA.

The Phase Two procedure MOVE.CARD stores these number pairs into PROGRAM.

PAIRS - PAIRS is the counter on the PROGRAM array. Its value at any time represents the number of pairs of library card numbers which have been stored into PROGRAM. Each time a pair of numbers is stored, PAIRS is incremented by one.

ARRAYBASE - The ARRAYBASE is the lowest number which can be used to represent a read-in array. It has been arbitrarily set at 10,000. Each time an array is recognized on an input data card by the AR procedure in Phase One, an identifying number for the array is placed into the current position of the STACK. The identifying numbers are equal to the ARRAYBASE plus the array number. Thus, any number in the STACK greater in magnitude than the ARRAYBASE but less than the PROCBASE is known to represent a read-in array. It should be noted that read-in single variables have a base of zero, and thus begin at one. Therefore, any number between zero and the ARRAYBASE represents a read-in single variable.

PROCBASE - The PROCBASE is the smallest number which can be used to represent a procedure. It is set in the compiler to 20,000. When Phase One recognizes a procedure call by means of the PR procedure, the identifying number for the procedure is placed into the current STACK position. These identifying numbers are equal to the PROCBASE plus the number of the procedure. Thus, any number in the STACK greater than the PROCBASE but less than the SPECBASE is known to represent a procedure.

SPECBASE - The SPECBASE, which has been set at 30,000, is the smallest number which can be used to represent a global variable, not of the read-in type. Such a variable, defined by the operation of procedures rather than read-in statements, may be either single or array. When it is known that a procedure defines a global variable of this type, the value of the SPECBASE plus the variable number is stored into the STACK at the current position of STORBASE by the LOAD.STACK procedure.

DATABLOCK - DATABLOCK tells the number of blocks of input data which have been written onto the SCRATCH AREA from the array DATABUFFER. It is always a multiple of 5, since 5 times 320 words/block, or 1600 words, are written at a time. These blocks of data, which are simply the input data cards, are written on the SCRATCH AREA by Phase One, and are later repositioned there to the end of the ALGOL program by Phase Three.

ERROR - The Boolean variable, ERROR, is used in order to keep the compiler from LINKing to its next compiler Phase if an error has been encountered in the Phase being executed. As long as no errors are found, ERROR remains FALSE; it is set to TRUE when an error is discovered.

Phase One Variables

IMAGE [1:81]
IMAG [1:31]
DATABUFFER [1:1620]
STNO
TYPE
SEGS
TRANSFER
SKIPOVER
DANO
EXCESS
STORED
HAVEPERIOD
TYPEDATA

4.2.4

IMAGE [1:81] - IMAGE is a logic array into which the contents of the 80 columns of each HYDRO input card are stored alphanumerically. This image of the input card is then scanned column by column in order to recognize its contents and place numbers into the STACK where appropriate.

IMAG [1:31] - When a variable name or procedure call is isolated on a card by Phase One, the letters which comprise it are transferred from IMAGE to IMAG. The contents of IMAG are then examined by the PR, SR, and AR procedures.

DATABUFFER [1:1620] - The DATABUFFER is a logic array into which data card images are stored prior to transferral to the end of the AND SCRATCH AREA. Each card image occupies 21 elements of the DATABUFFER. When a data card is recognized, its image is placed into the DATABUFFER, and when the buffer is filled beyond 1600 words, it is transferred onto the SCRATCH AREA.

STNO - The stack number, STNO, is the upper pointer on the STACK array during Phase One. It gives the next position in the STACK into which a number representing a procedure or data variable will be placed. STNO is initially set at 500, and is incremented by one each time a number is stored into the STACK.

TYPE - TYPE can have the values of 0 or 1. It is used to indicate which type of HYDRO input is currently being scanned in IMAGE. If TYPE is zero, a procedure has been scanned. TYPE is set to one when a data variable is scanned.

SEGS - SEGS is an integer variable containing the number of segments shich comprise Phase One. It is used as a check on the TRANSFER variable when a procedure or variable is being recognized by calls on PR, SR, and AR.

TRANSFER - TRANSFER is an integer counter used when an attempt is being made to match a procedure or variable name in IMAG with a call on PR, SR, or AR. It is incremented by one each time a transfer is made to a new segment of Phase One in an attempt to made the match. It is not allowed to go beyond SEGS, and is reset to one each time a match is made.

SKIPOVER - This Boolean variable is set to TRUE when a procedure or variable name has been located in IMAGE and stored into IMAG, and a TRANSFER is about to be made to another segment of Phase One. It prevents the new segment from re-executing the code in which the procedure or variable name was scanned and stored into IMAG.

DANO - The data number, DANO, is the pointer on the DATABUFFER array. DANO begins at one, and each time a 4-character logic word is stored into the DATABUFFER it is incremented by one. When DANO exceeds 1600, the DATABUFFER is emptied, and DANO is reset to one. DANO always represents the position in the DATABUFFER into which the next word is to be read from the input data cards.

EXCESS - EXCESS is used in Phase One in the section of code where data card images are transferred from the DATABUFFER to the SCRATCH AREA. EXCESS tells at the time of transferral how many logic words have been placed into DATABUFFER beyond 1600. When 1600 words are transferred to the SCRATCH AREA, this EXCESS is placed at the beginning of the DATABUFFER so that it will be included the next time the DATABUFFER is emptied.

STORED - STORED is a Boolean variable which indicates whether or not the image of a data card has been placed into the DATABUFFER. If STORED is FALSE, it indicates that the card in IMAGE has not been placed into the DATABUFFER. As soon as it is placed there, STORED becomes TRUE.

HAVEPERIOD - HAVEPERIOD is a Boolean variable which tells whether or not a period has been located in a piece of numerical data being scanned on a data card. Before each number is scanned, HAVEPERIOD is set to FALSE, and is made TRUE whenever a decimal point is found. Its purpose is to prevent the occurrence of more than one period in any piece of numerical data.

TYPEDATA - The logic variable, TYPEDATA, is used to indicate what type of data, numeric or alphanumeric, is currently being scanned on a data card. If numerical data is located on a data card, TYPEDATA is set to NUMR, whereas if logic data is being scanned, it is set to ALPH. The purpose of TYPEDATA is to prevent both kinds of data from being stored into the same variable.

Phase Two Variables

BASE1
BASE2
BASE3
BASE4
OPNO
STORBASE
DATAREAD
DECLARED
TEMP1
TEMP2

BASE1 - BASE1 is the number of the card image on the language library, AND RECORD TWO, where the declarations of global single variables end, and the data read statements for single variables begin. BASE1 is used in the SINGLE.VARIABLE procedure to compute the parameters of calls on MOVE.CARD for the single variable read cards.

BASE2 - The value of BASE2 represents the library card image which separates the data read statements for single variables from those for arrays. It is used in the ARRAY.VARIABLE procedure to compute the parameters of the MOVE.CARD calls for array read code.

BASE3 - BASE3 represents the library card image which separates the array reads from the material which is declared global to each of the procedures. It is used in calls on MOVE.CARD for sections of this global material.

BASE4 - The library card image which separates the material global to procedures from the procedure bodies is given by the value of BASE4. This variable is used in the calls on MOVE.CARD for the bodies of the procedures and the procedure calls.

OPNO - The operation number, OPNO, is the upper pointer of the STACK during Phase Two, and tells what element of the STACK is being processed. The OPNO begins at the same place as STORBASE, the middle element of the STACK, but it is incremented while STORBASE is decremented.

STORBASE - The STORBASE is the lower pointer of the STACK, used during Phase Two. It begins at 500 and is decremented by one each time a number is placed into the STACK representing a global variable defined by one of the procedures.

DATAREAD - DATAREAD is a Boolean variable which tells at any time whether or not MOVE.CARD has been called for the special data reading procedures which should be placed at the head of the final ALGOL program being assembled on the SCRATCH AREA. This variable is set FALSE at the exit to Phase One so that immediately upon entry to Phase Two, MOVE.CARD will be called for these procedures.

DATAREAD is then made TRUE, and the procedures are never assembled again into the same ALGOL program.

DECLARED - DECLARED is a Boolean variable used in the ARRAY.VARIABLE procedure. Its TRUE or FALSE value indicates whether or not the array variable being processed has ever been declared previously in the ALGOL program to be assembled on the SCRATCH AREA.

TEMP1 and TEMP2 - TEMP1 and TEMP2 are temporary storage locations used in Phase Two. When a number representing a procedure is found in the STACK, its position in the STACK is temporarily stored into TEMP1, and the number itself is stored into TEMP2. The STACK element is then set to zero. This enables the IN.THE.STACK call for the procedure, following the PROCESS.VARIABLES call, to give correct results. After complete processing of the procedure, the original procedure number is returned to the STACK.

Phase Three Variables

READBUFFER [1:1621]
WRITEBUFFER [1:1600]
LOWORD
HIWORD
UPPERBOUND
FIRSTWORD
LASTWORD
READPOINT
WRITEPOINT
READBLOCK
WRITEBLOCK
WORDNO

READBUFFER [1:1621] - The READBUFFER is a logic vector, 1621 words in length, used in the TRANSFER.CARD procedure. Five blocks of code at a time are read alphanumerically into this buffer by the DISCREAD procedure. Some or all of this code may then be stored into the WRITEBUFFER prior to transferral to the SCRATCH AREA.

WRITEBUFFER [1:1600] - The WRITEBUFFER is a logic vector of 1600 words. It is loaded with sections of material from the READBUFFER, and when filled to 1600 words, is written onto the SCRATCH AREA by the DISCWRITE routine.

LOWORD - The value of LOWORD indicates the numerical position of the first four-character word which is to be transferred from the language library to the SCRATCH AREA in a call on TRANSFER.CARD.

HIWORD - HIWORD represents the last four-character word that is to be transferred from the library to the SCRATCH AREA in any call on TRANSFER.CARD.

UPPERBOUND - The UPPERBOUND is the numerical position of the last four-character alphanumeric word read from the library into the READBUFFER. It is always a multiple of 1600, since 1600 words are read at a time.

FIRSTWORD - The value of FIRSTWORD indicates the position in the 1600 words of the READBUFFER where transferral is to begin from the READBUFFER to the WRITEBUFFER in a call on TRANSFER.BUFFER.

LASTWORD - This variable tells the last word of the READBUFFER which is to be transferred to the WRITEBUFFER in a call on TRANSFER.BUFFER.

READPOINT - READPOINT is the counter on the READBUFFER. During the TRANSFER.BUFFER procedure it is initially set to FIRSTWORD, and is then incremented in steps of one during an iteration process up to the value of LASTWORD.

WRITEPOINT - The WRITEPOINT is the counter on the WRITEBUFFER. It begins at zero and is incremented in steps of one each time a word is placed into WRITEBUFFER until it reaches a final value of 1600. When the WRITEBUFFER is emptied it is reset to zero.

READBLOCK - The value of READBLOCK represents the block of the library which was the last to be read into READBUFFER.

WRITEBLOCK - The WRITEBLOCK indicates the number of blocks of code which have been written onto the SCRATCH AREA from the WRITEBUFFER.

WORDNO - The variable WORDNO tells at any time during Phase Three the number of words which have been written onto the SCRATCH AREA by the DISCWRITE routine. It is used at the end of Phase Three in a calculation to find the number of blank words which must be placed into the final program line on the SCRATCH AREA, in order that the data card images can be transferred correctly from the end of the SCRATCH AREA.

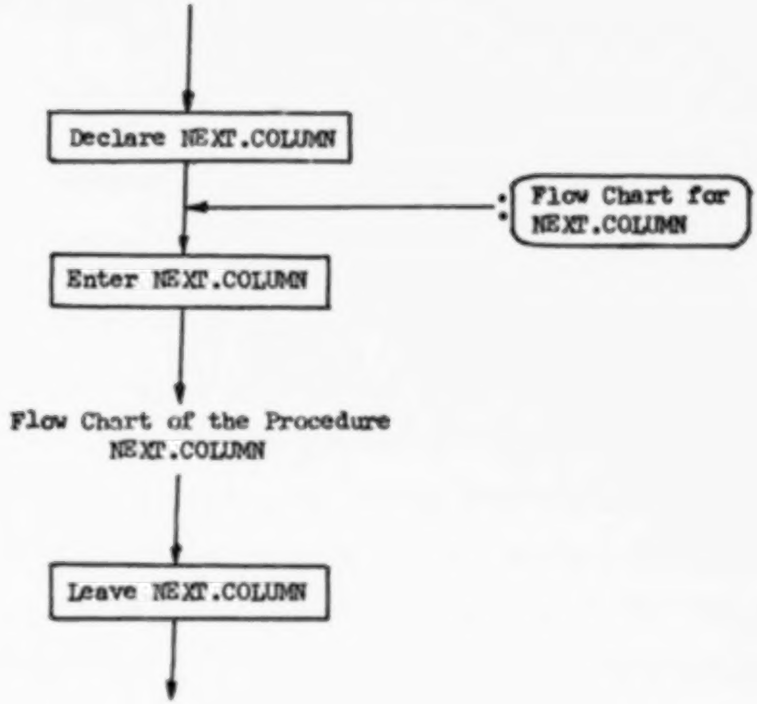
5. DETAILED COMPILER FLOW CHART

A detailed flow chart of the HYDRO compiler is presented in this Chapter, followed by its explanation. This chart is detailed to the point that each box represents one or two lines of code in the actual programs. Therefore, this Chapter should be read only after a working knowledge of the compiler has been obtained from Chapter 2 dealing with the compiler as a component of the total HYDRO system, and from Chapter 3, which presents and discusses a much more general flow chart of the compiler.

The flow chart in this Chapter should be studied in conjunction with the program listings of the compiler presented in Appendix II, so that each box of the flow chart can be matched with the corresponding code in the actual compiler programs.

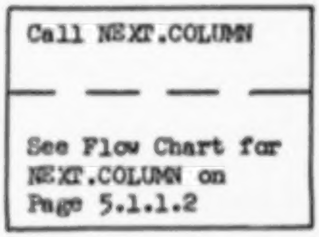
Embedded within the flow chart are many smaller charts for the individual procedures which form parts of the different compiler Phases. These procedure flow charts will be presented in the general flow chart of the compiler in a position corresponding to the declaration of the procedures in the actual programs.

The format for presenting these flow charts will be identical for each procedure, and is illustrated below for the procedure NEXT.COLUMN.


Declare NEXT.COLUMN
Flow Chart for NEXT.COLUMN
Enter NEXT.COLUMN
Flow Chart of the Procedure NEXT.COLUMN
Leave NEXT.COLUMN

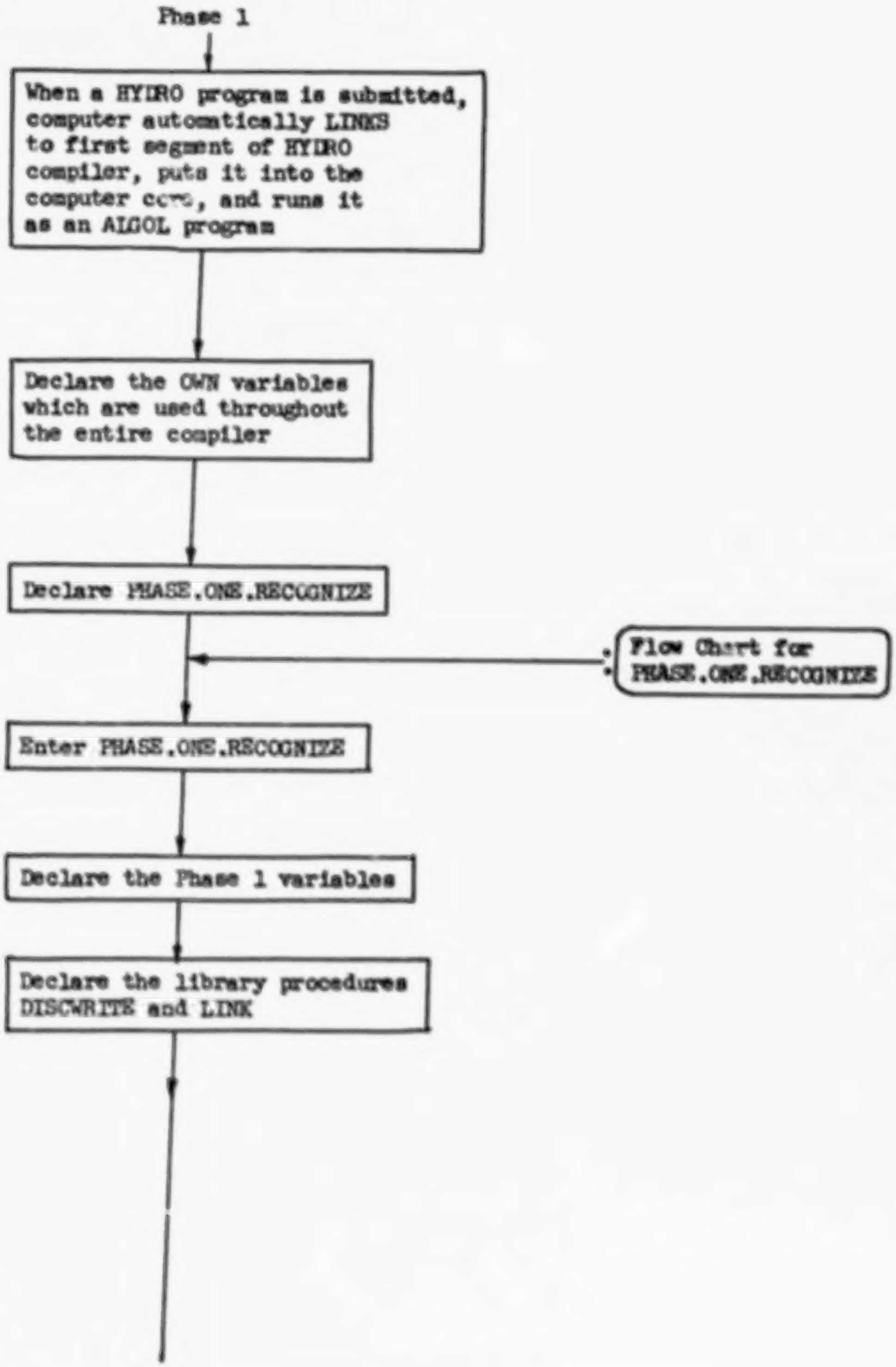
The first box in this diagram indicates the place where the procedure is declared in the general flow chart. This introduces the flow chart for the individual procedure (NEXT.COLUMN) which begins with a label designator indicating that the flow chart for the procedure follows. The flow chart for the procedure is bounded by two boxes indicating where to enter and leave the procedure.

Later in the general flow chart calls are made on procedures previously declared. When this occurs, a reference is made to the page where the flow chart of that procedure is presented, similar to the following:

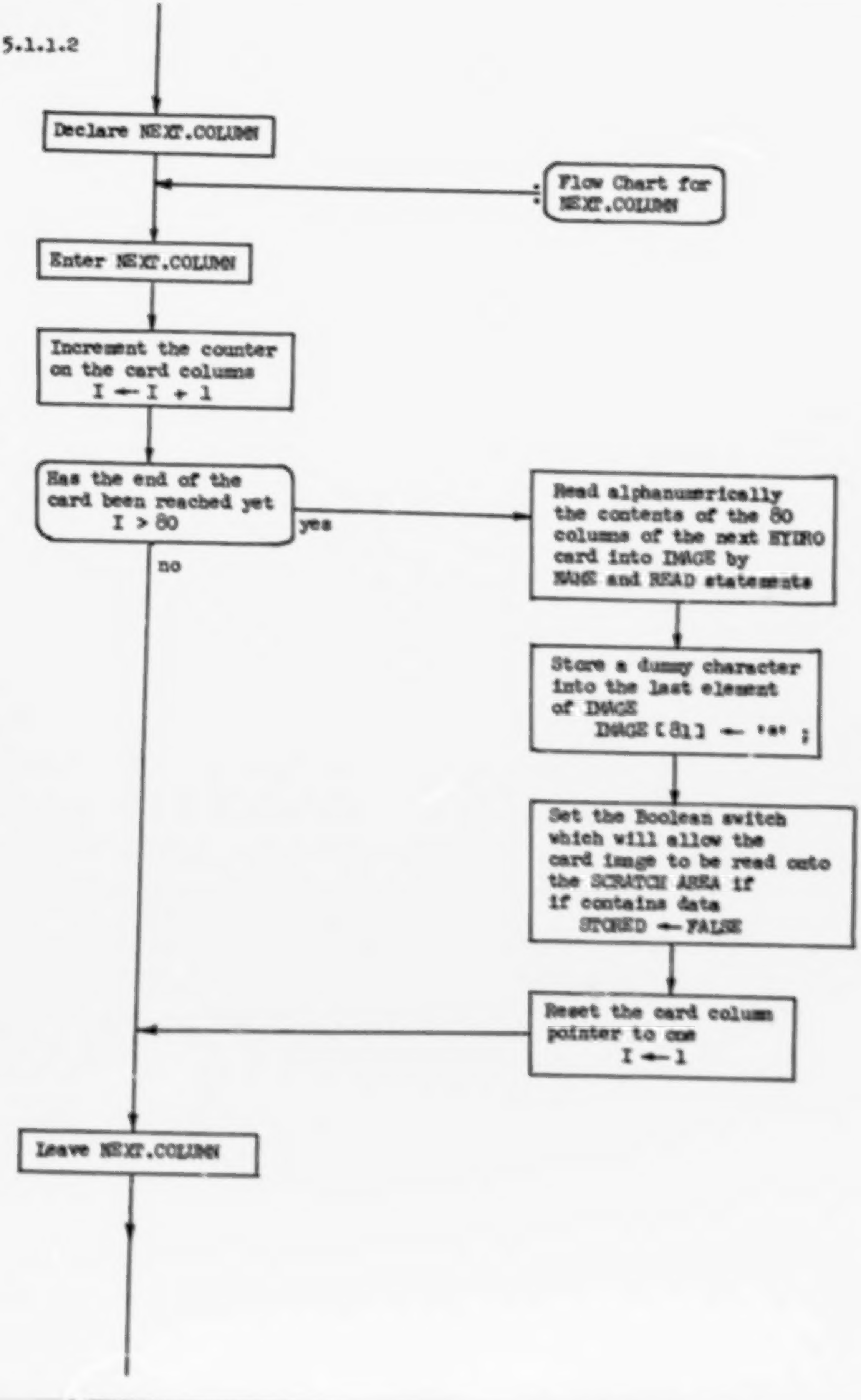

Call NEXT.COLUMN
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2

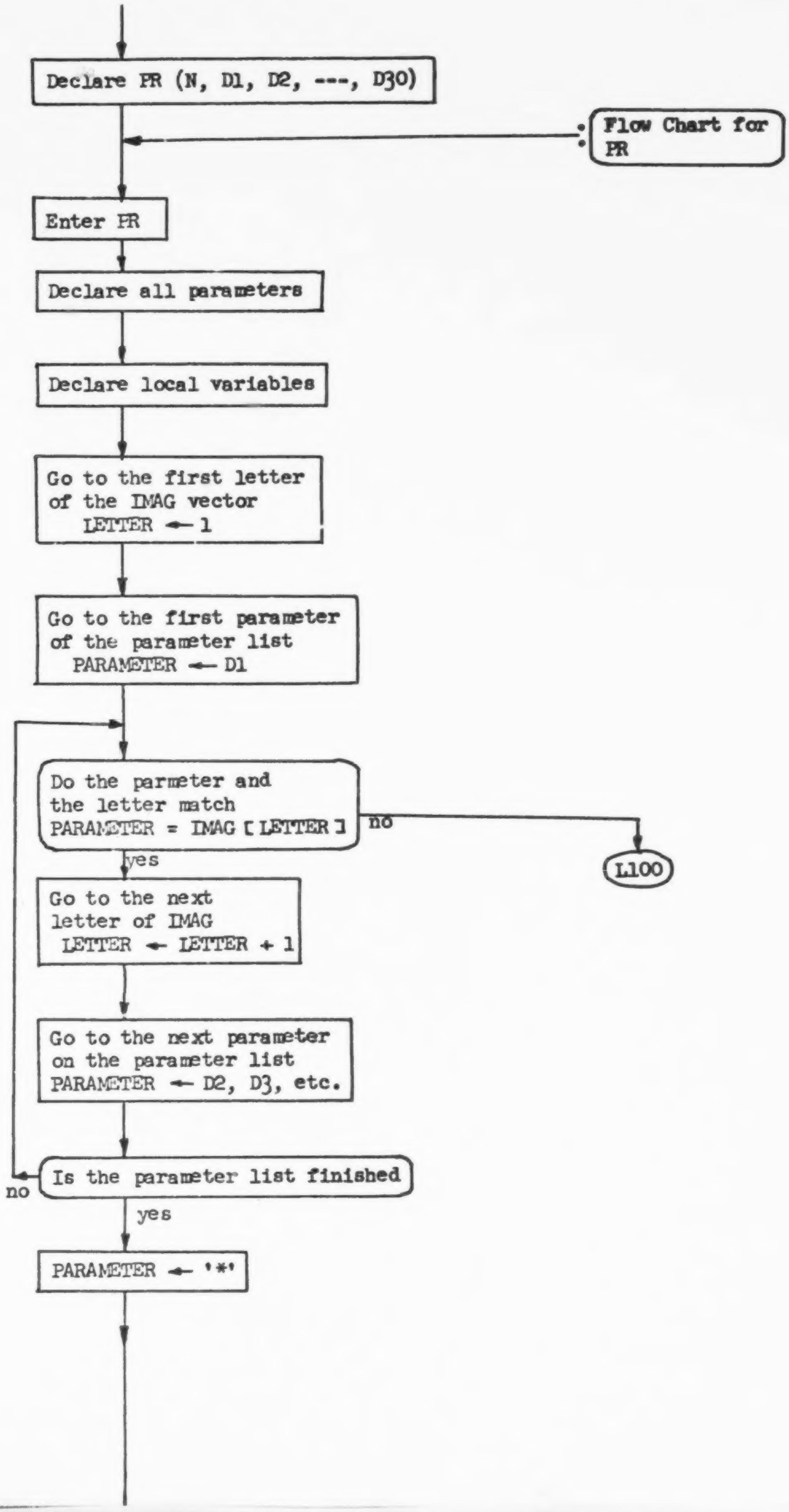
5.1 Flow Chart

5.1.1 Phase One - Recognition

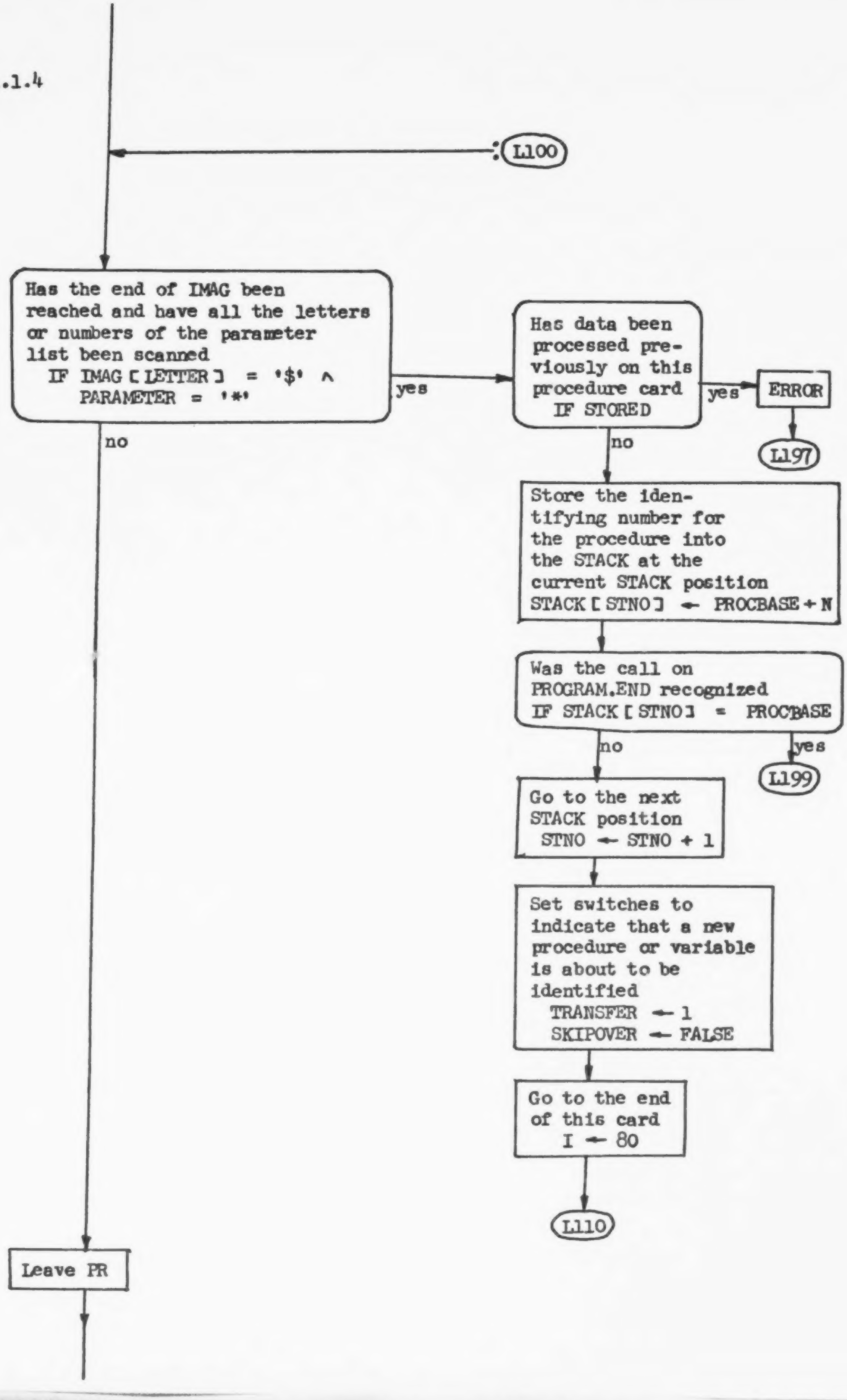

Phase 1
When a HYDRO program is submitted, computer automatically LINKS to first segment of HYDRO compiler, puts it into the computer core, and runs it as an ALGOL program
Declare the OWN variables which are used throughout the entire compiler
Declare PHASE.ONE.RECOGNIZE
Flow Chart for PHASE.ONE.RECOGNIZE
Enter PHASE.ONE.RECOGNIZE
Declare the Phase 1 variables
Declare the library procedures DISCWRITE and LINK

5.1.1.2

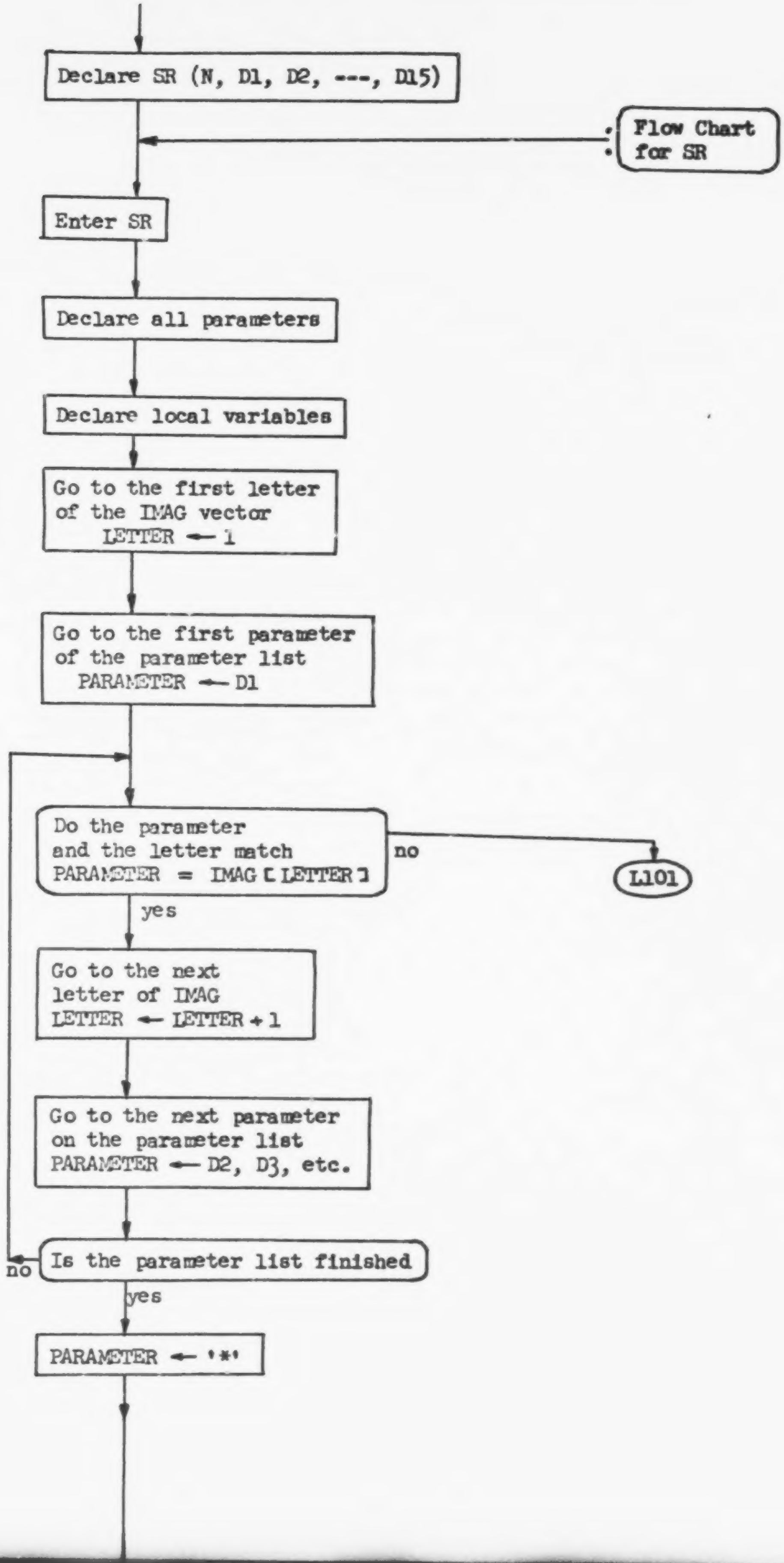

Declare NEXT.COLUMN
Flow Chart for NEXT.COLUMN
Enter NEXT.COLUMN
Increment the counter on the card columns I ← I + 1
Has the end of the card been reached yet I > 80
yes
no
Read alphanumerically the contents of the 80 columns of the next HYDRO card into IMAGE by NAME and READ statements
Store a dummy character into the last element of IMAGE IMAGE [81] ← '*' ;
Set the Boolean switch which will allow the card image to be read onto the SCRATCH AREA if if contains data STORED ← FALSE
Reset the card column pointer to one I ← 1
Leave NEXT.COLUMN


Declare PR (N, D1, D2, ---, D30)
Flow Chart for PR
Enter PR
Declare all parameters
Declare local variables
Go to the first letter of the IMAG vector LETTER ← 1
Go to the first parameter of the parameter list PARAMETER ← D1
Do the parmeter and the letter match PARAMETER = IMAG [LETTER]
no
L100
yes
Go to the next letter of IMAG LETTER ← LETTER + 1
Go to the next parameter on the parameter list PARAMETER ← D2, D3, etc.
Is the parameter list finished
no
yes
PARAMETER ← '*'

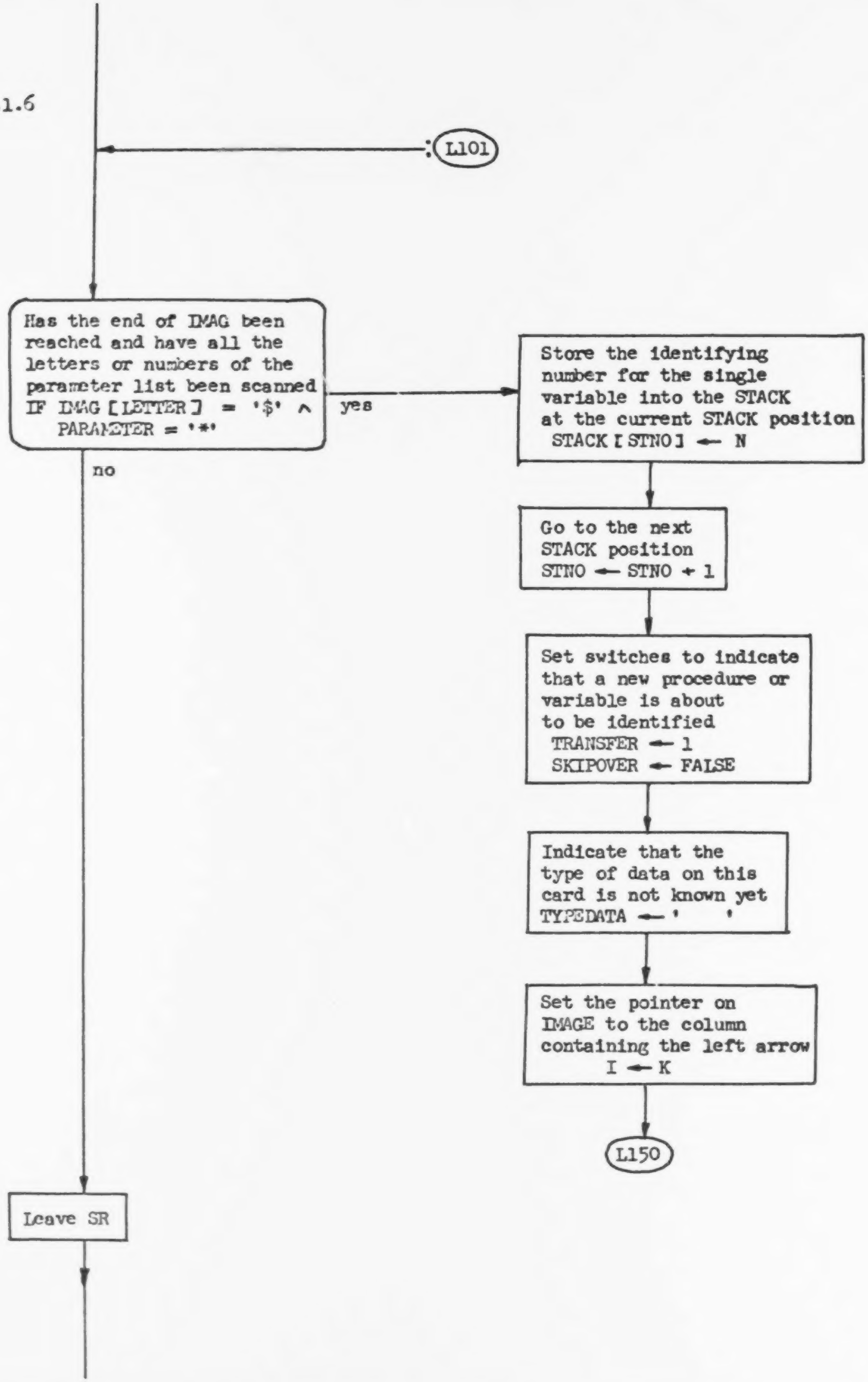
5.1.1.4

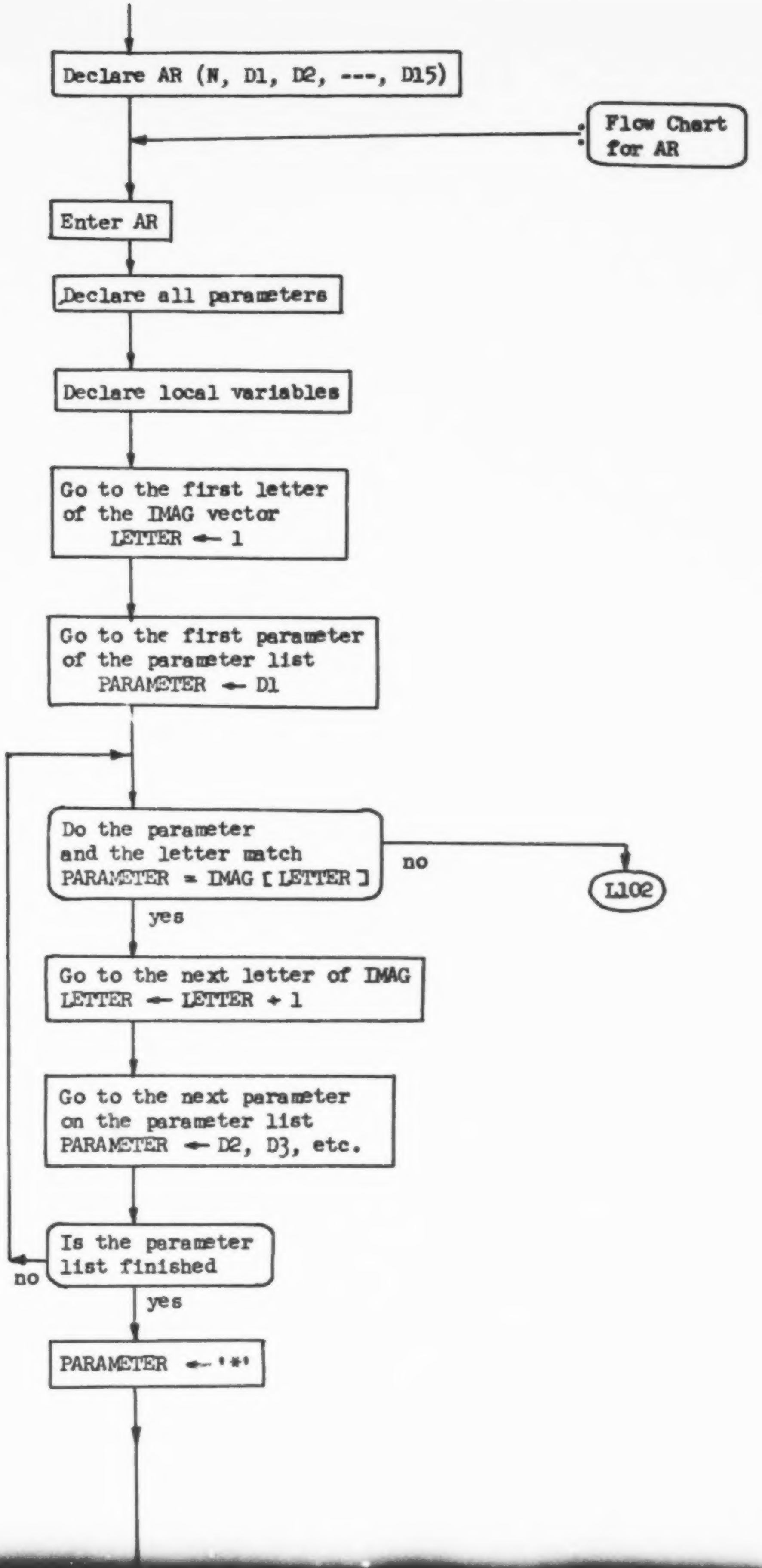

L100
Has the end of IMAG been reached and have all the letters or numbers of the parameter list been scanned
IF IMAG [LETTER] = '$' ∧ PARAMETER = '*'
yes
no
Has data been processed previously on this procedure card
IF STORED
yes
ERROR
L197
no
Store the identifying number for the procedure into the STACK at the current STACK position
STACK [STNO] ← PROCBASE + N
Was the call on PROGRAM.END recognized
IF STACK [STNO] = PROCBASE
no
yes
L199
Go to the next STACK position
STNO ← STNO + 1
Set switches to indicate that a new procedure or variable is about to be identified
TRANSFER ← 1
SKIPOVER ← FALSE
Go to the end of this card
I ← 80
L110
Leave PR

5.1.1.5

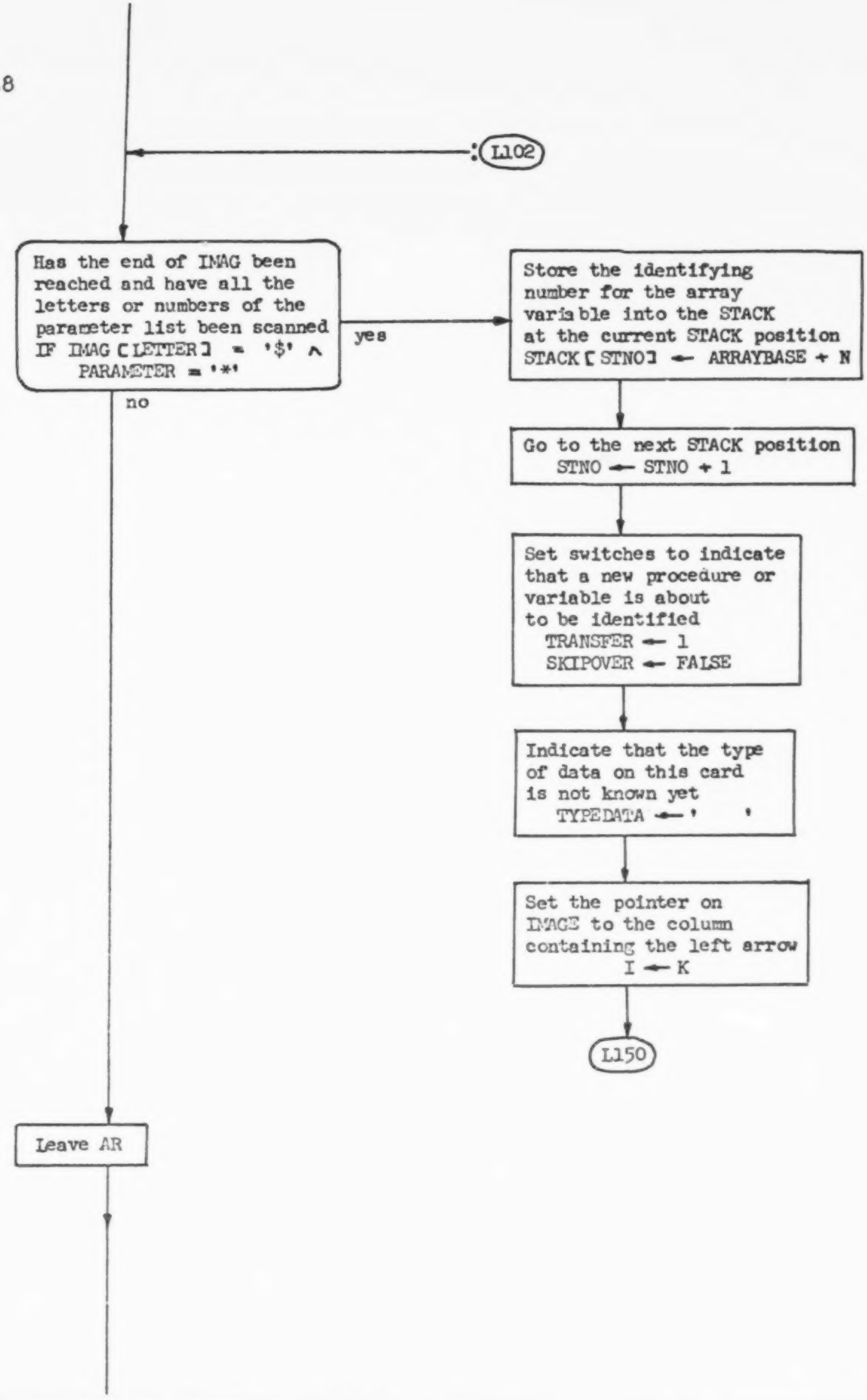

Declare SR (N, D1, D2, ---, D15)
Flow Chart for SR
Enter SR
Declare all parameters
Declare local variables
Go to the first letter of the IMAG vector
LETTER ← 1
Go to the first parameter of the parameter list
PARAMETER ← D1
Do the parameter and the letter match
PARAMETER = IMAG [LETTER]
no
L101
yes
Go to the next letter of IMAG
LETTER ← LETTER + 1
Go to the next parameter on the parameter list
PARAMETER ← D2, D3, etc.
no
Is the parameter list finished
yes
PARAMETER ← '*'

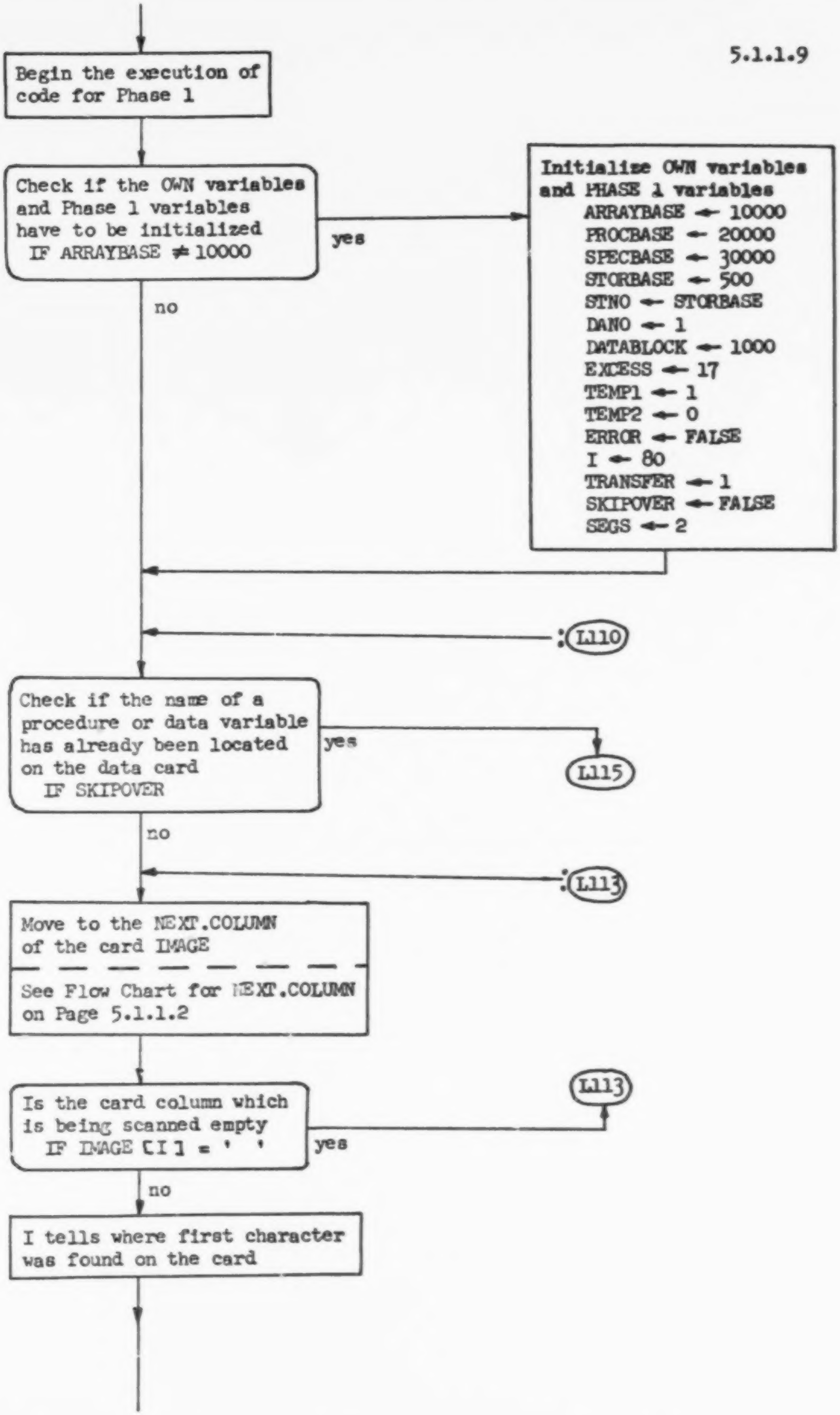
5.1.1.6


L101
Has the end of IMAG been reached and have all the letters or numbers of the parameter list been scanned
IF IMAG [LETTER] = '$' ∧ PARAMETER = '*'
yes
no
Store the identifying number for the single variable into the STACK at the current STACK position
STACK [STNO] ← N
Go to the next STACK position
STNO ← STNO + 1
Set switches to indicate that a new procedure or variable is about to be identified
TRANSFER ← 1
SKIPOVER ← FALSE
Indicate that the type of data on this card is not known yet
TYPEDATA ← '   '
Set the pointer on IMAGE to the column containing the left arrow
I ← K
L150
Leave SR

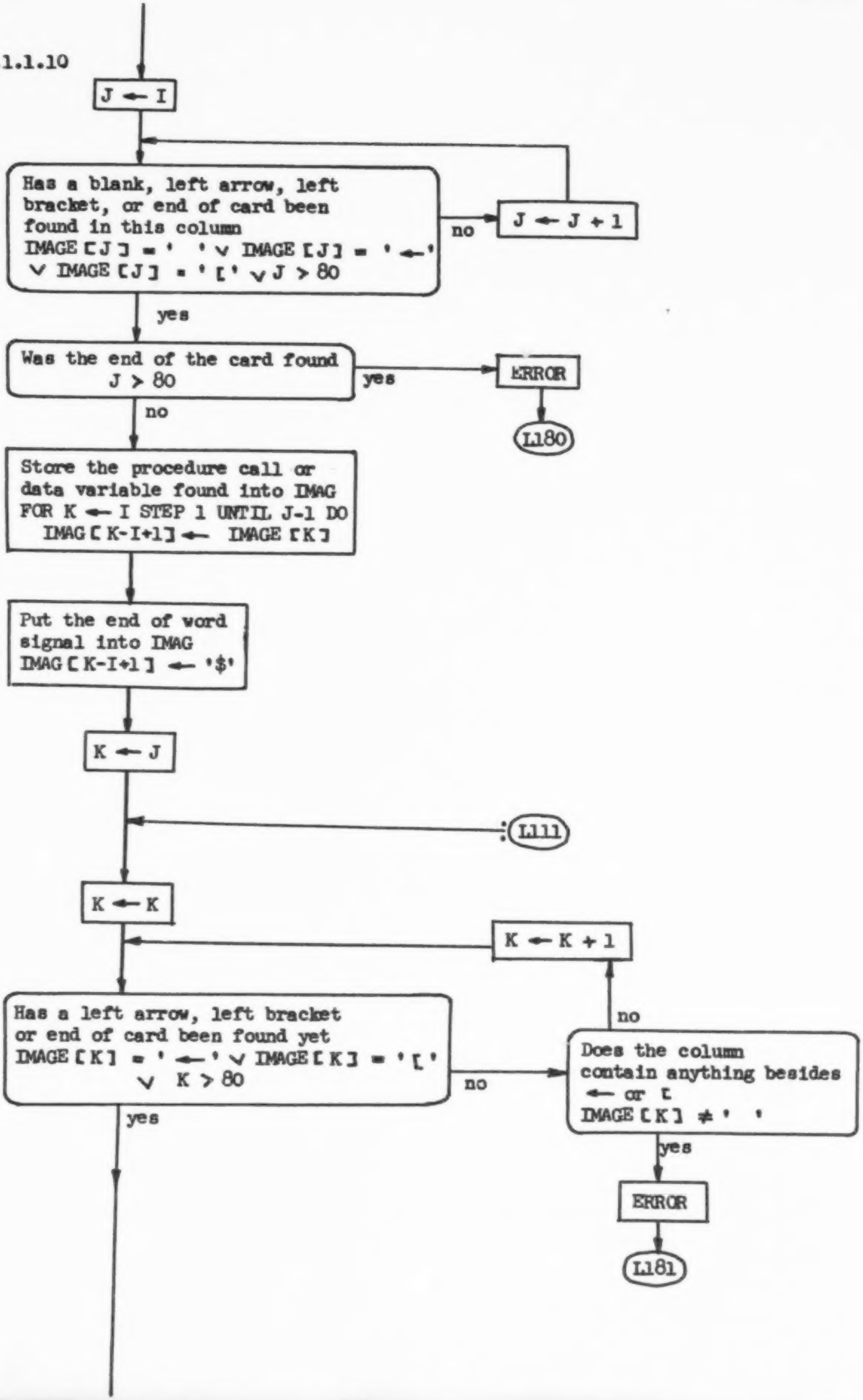

Declare AR (N, D1, D2, ---, D15)
Flow Chart for AR
Enter AR
Declare all parameters
Declare local variables
Go to the first letter of the IMAG vector
LETTER ← 1
Go to the first parameter of the parameter list
PARAMETER ← D1
Do the parameter and the letter match
PARAMETER = IMAG [LETTER]
no
L102
yes
Go to the next letter of IMAG
LETTER ← LETTER + 1
Go to the next parameter on the parameter list
PARAMETER ← D2, D3, etc.
Is the parameter list finished
no
yes
PARAMETER ← '*'

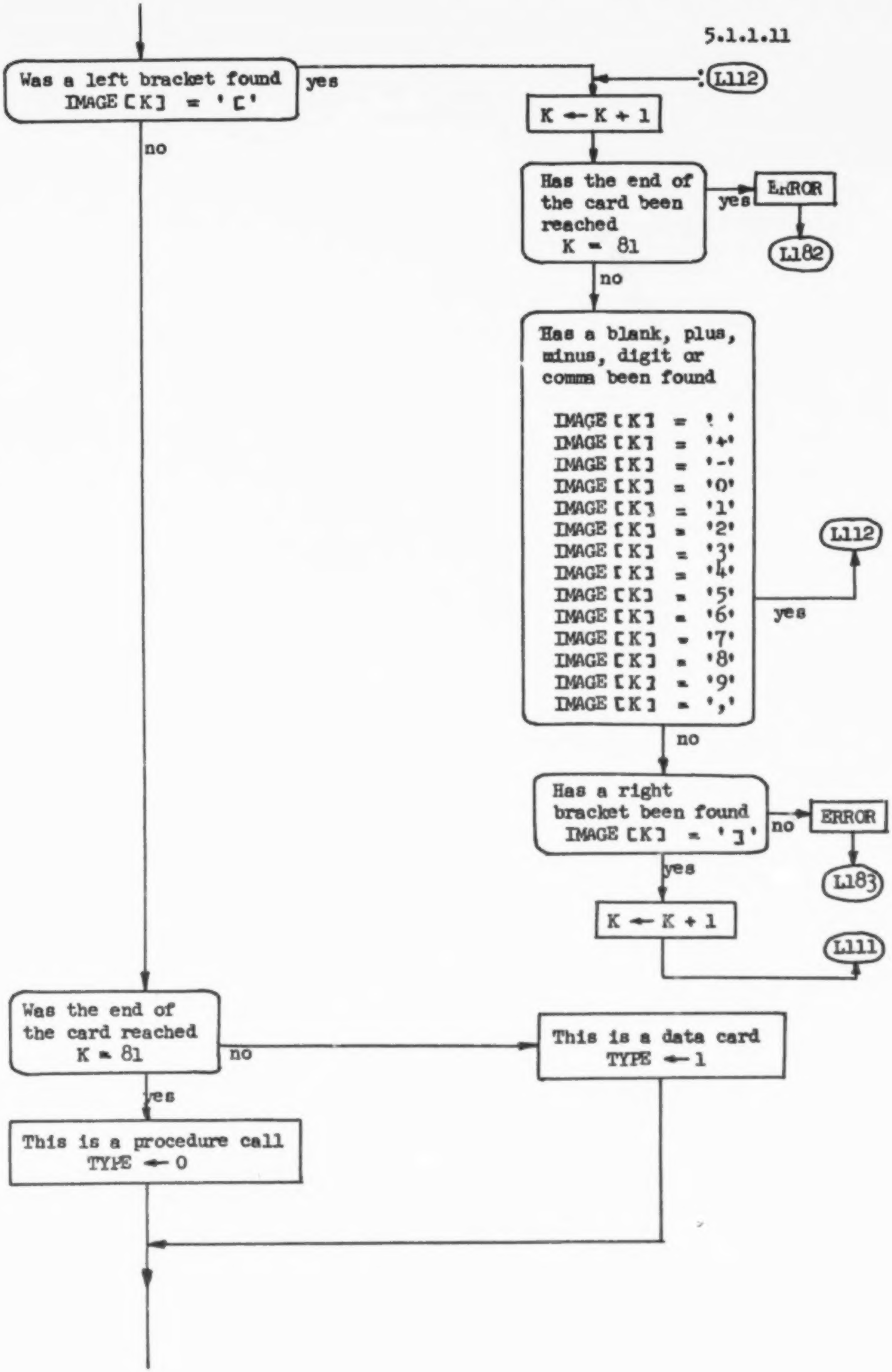
5.1.1.8

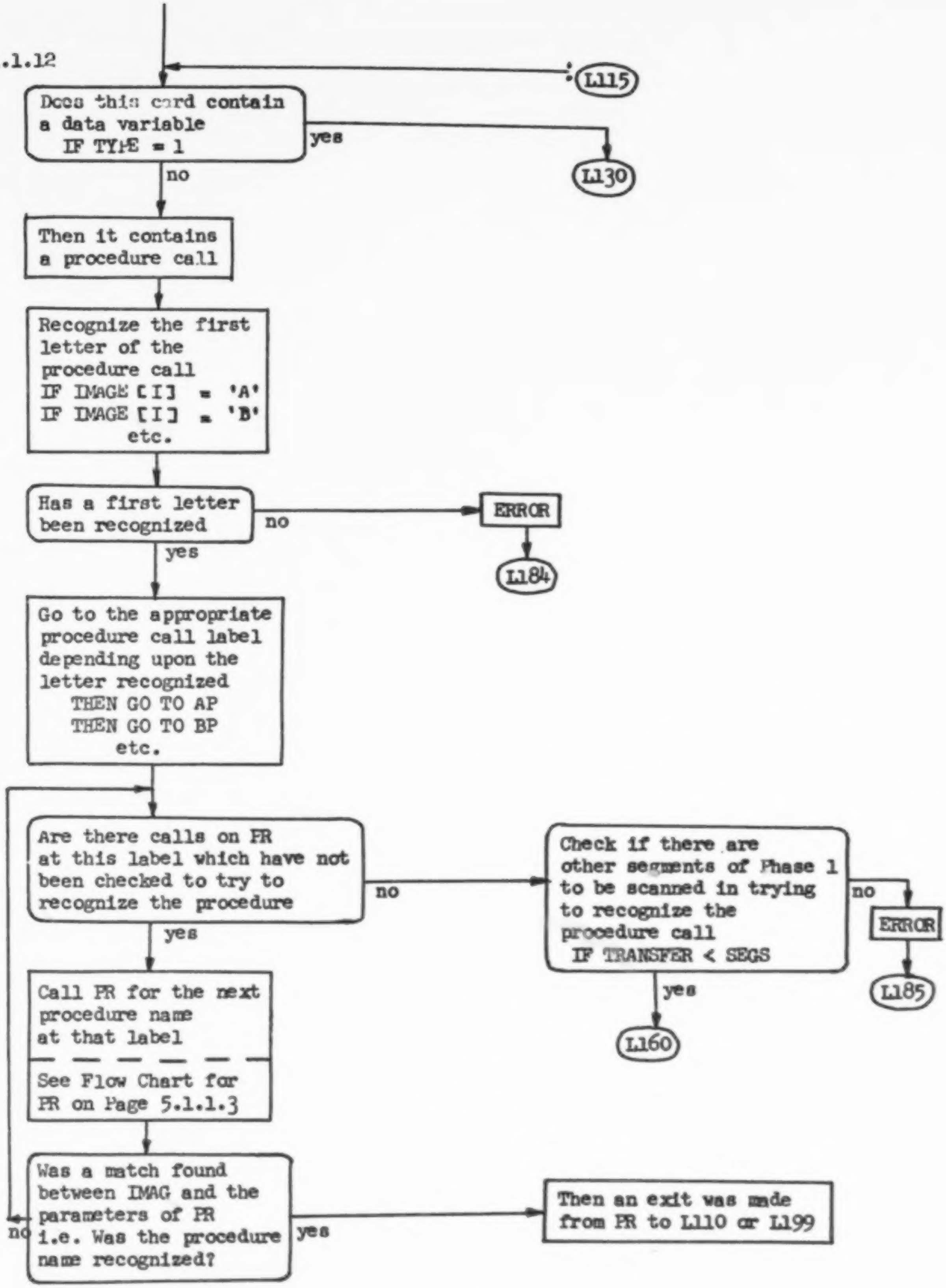

L102
Has the end of IMAG been reached and have all the letters or numbers of the parameter list been scanned
IF IMAG [LETTER] = '$' ∧ PARAMETER = '*'
yes
no
Store the identifying number for the array variable into the STACK at the current STACK position
STACK [STNO] ← ARRAYBASE + N
Go to the next STACK position
STNO ← STNO + 1
Set switches to indicate that a new procedure or variable is about to be identified
TRANSFER ← 1
SKIPOVER ← FALSE
Indicate that the type of data on this card is not known yet
TYPEDATA ← ' '
Set the pointer on IMAGE to the column containing the left arrow
I ← K
L150
Leave AR

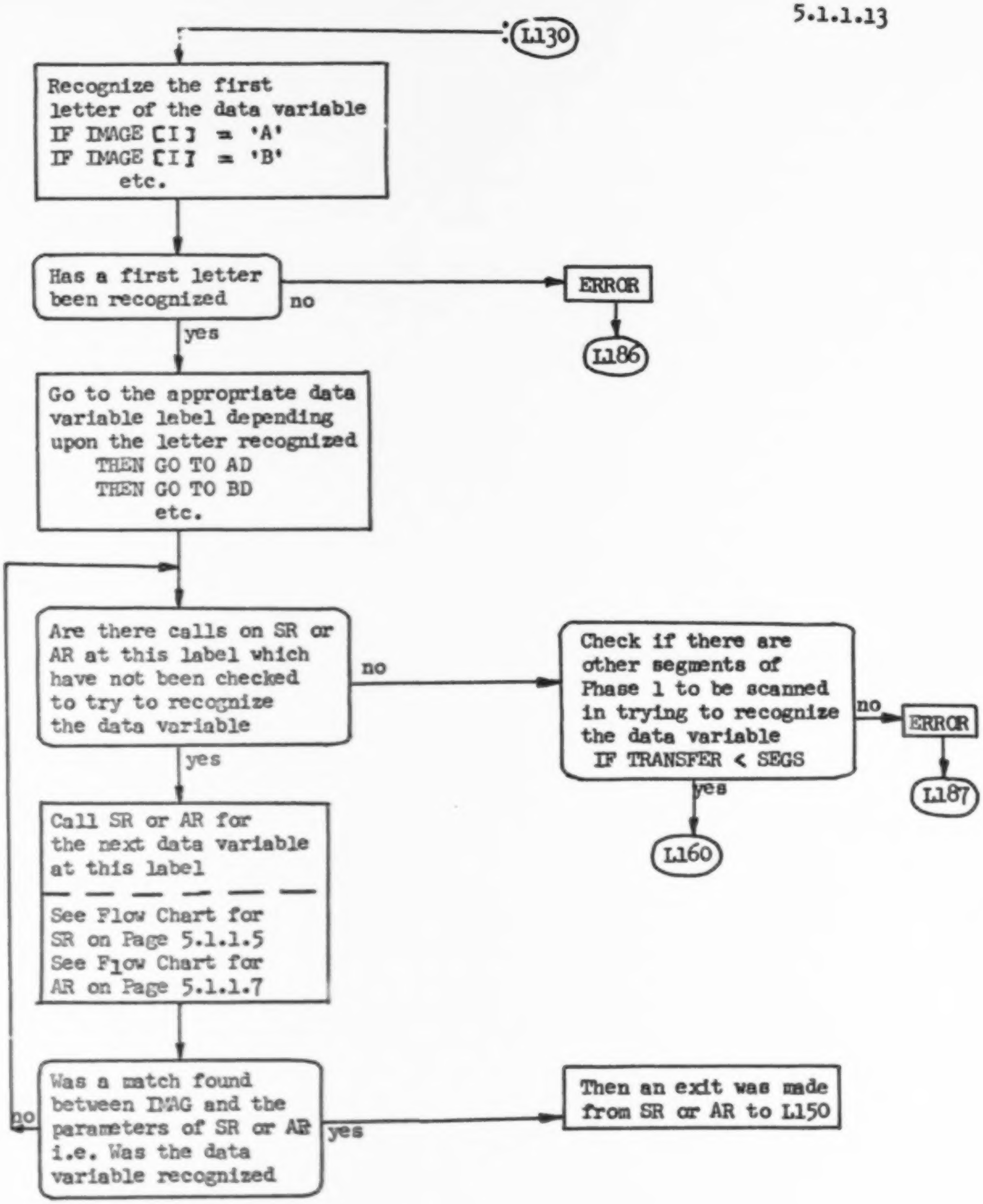

Begin the execution of code for Phase 1
Check if the OWN variables and Phase 1 variables have to be initialized IF ARRAYBASE ≠ 10000
yes
no
Initialize OWN variables and PHASE 1 variables
ARRAYBASE ← 10000
PROCBASE ← 20000
SPECBASE ← 30000
STORBASE ← 500
STNO ← STORBASE
DANO ← 1
DATABLOCK ← 1000
EXCESS ← 17
TEMP1 ← 1
TEMP2 ← 0
ERROR ← FALSE
I ← 80
TRANSFER ← 1
SKIPOVER ← FALSE
SEGS ← 2
:L110
Check if the name of a procedure or data variable has already been located on the data card IF SKIPOVER
yes
L115
no
:L113
Move to the NEXT.COLUMN of the card IMAGE
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2
Is the card column which is being scanned empty IF IMAGE [I] = ' '
yes
L113
no
I tells where first character was found on the card

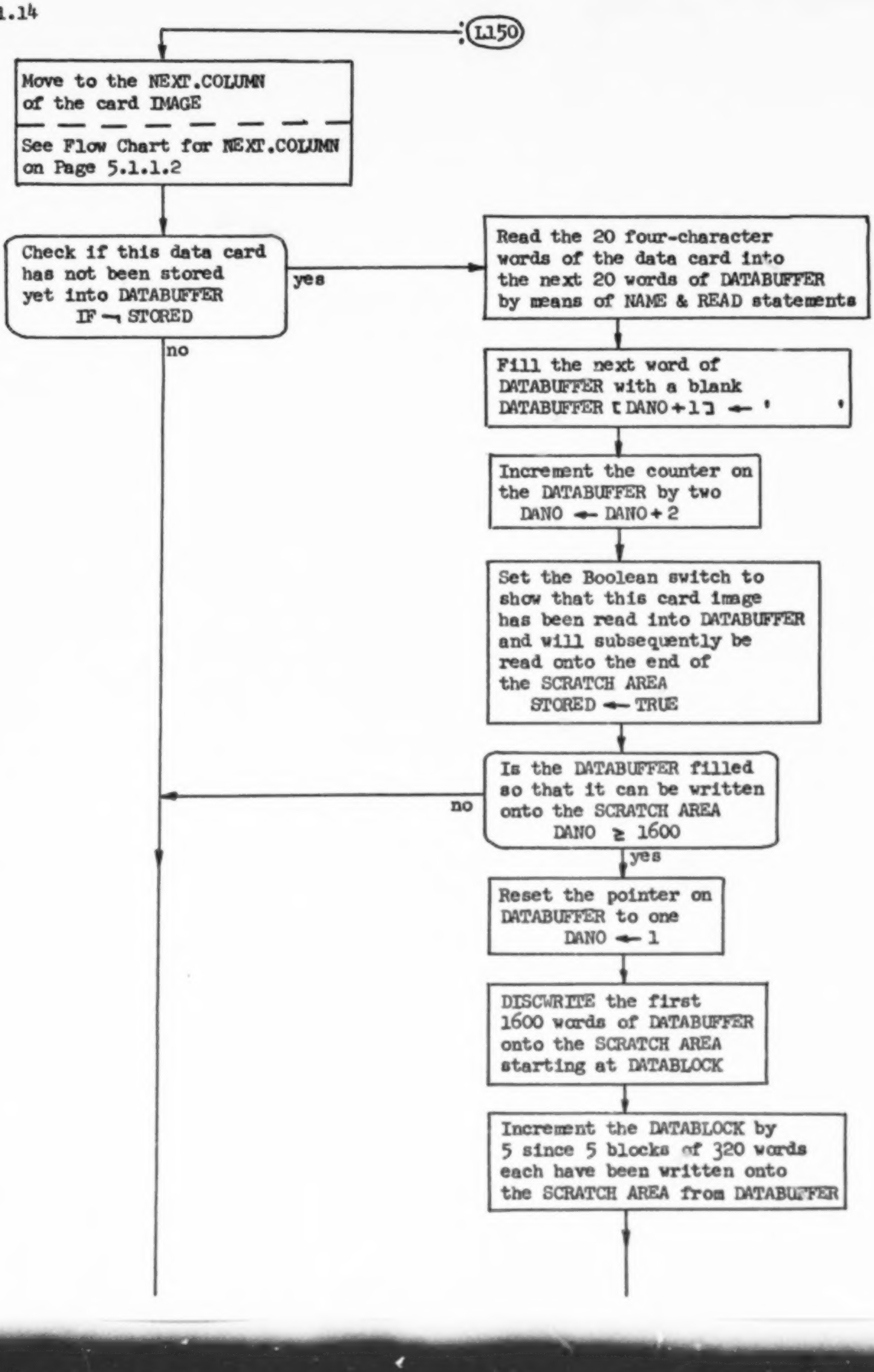
5.1.1.10

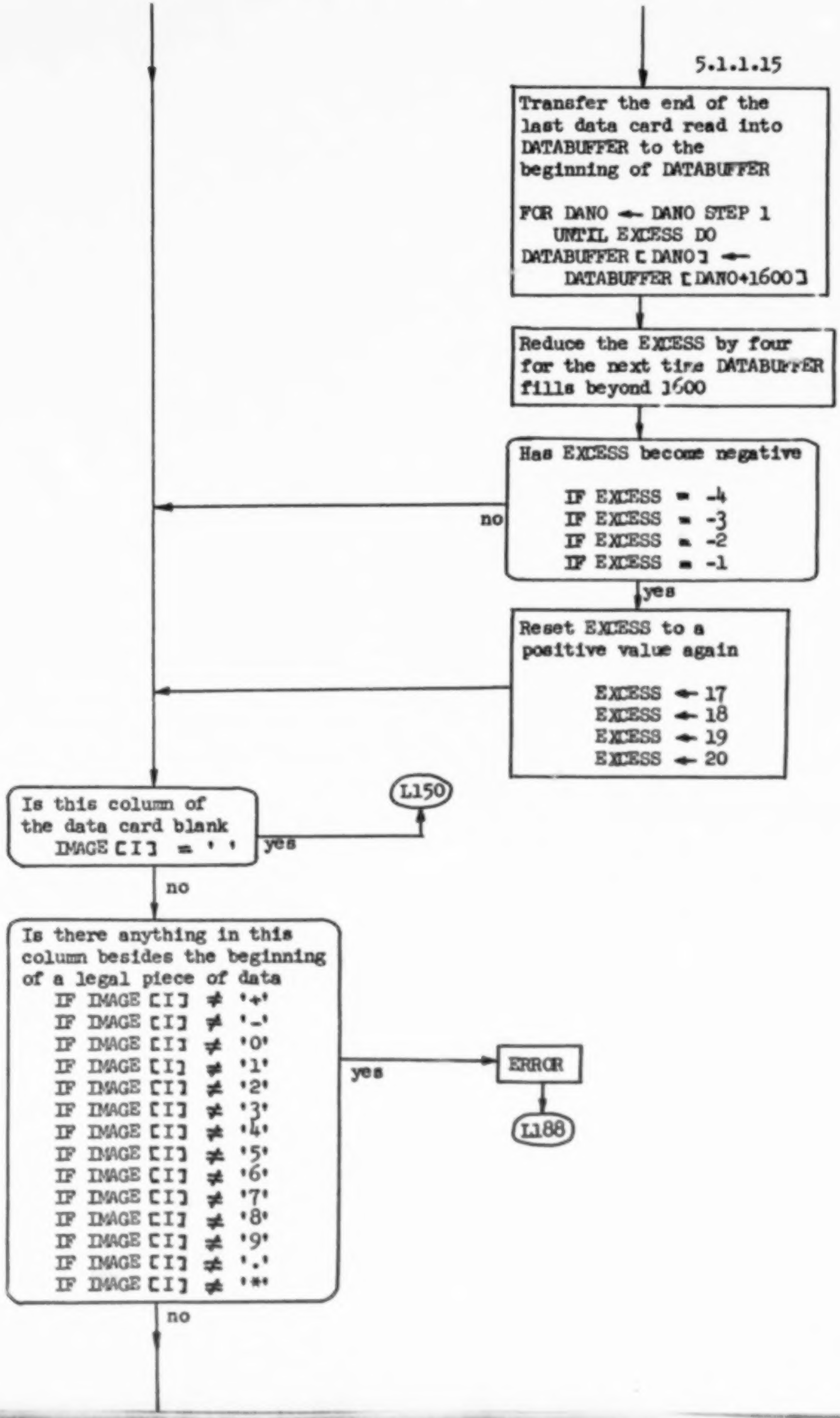

J ← I
Has a blank, left arrow, left bracket, or end of card been found in this column
IMAGE [J] = ' ' ∨ IMAGE [J] = '←' ∨ IMAGE [J] = '[' ∨ J > 80
no
J ← J + 1
yes
Was the end of the card found J > 80
yes
ERROR
L180
no
Store the procedure call or data variable found into IMAG
FOR K ← I STEP 1 UNTIL J-1 DO
IMAG [K-I+1] ← IMAGE [K]
Put the end of word signal into IMAG
IMAG [K-I+1] ← '$'
K ← J
L111
K ← K
K ← K + 1
Has a left arrow, left bracket or end of card been found yet
IMAGE [K] = '←' ∨ IMAGE [K] = '[' ∨ K > 80
no
no
Does the column contain anything besides ← or [
IMAGE [K] ≠ ' '
yes
ERROR
L181
yes

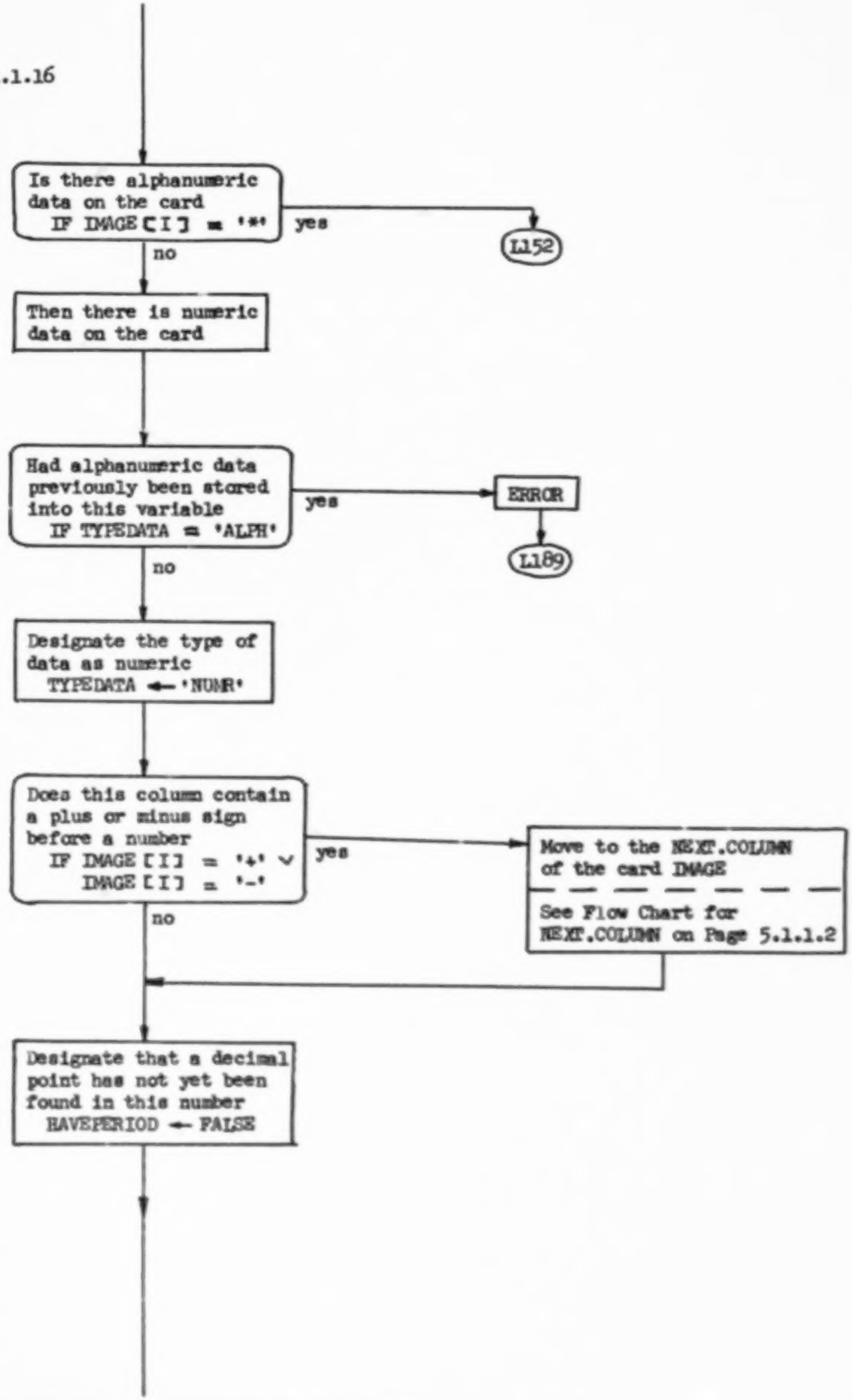

Was a left bracket found
IMAGE [K] = '['
yes
L112
K ← K + 1
Has the end of the card been reached
K = 81
yes
ERROR
L182
no
Has a blank, plus, minus, digit or comma been found
IMAGE [K] = ' '
IMAGE [K] = '+'
IMAGE [K] = '-'
IMAGE [K] = '0'
IMAGE [K] = '1'
IMAGE [K] = '2'
IMAGE [K] = '3'
IMAGE [K] = '4'
IMAGE [K] = '5'
IMAGE [K] = '6'
IMAGE [K] = '7'
IMAGE [K] = '8'
IMAGE [K] = '9'
IMAGE [K] = ','
yes
L112
no
Has a right bracket been found
IMAGE [K] = ']'
no
ERROR
L183
yes
K ← K + 1
L111
no
Was the end of the card reached
K = 81
no
This is a data card
TYPE ← 1
yes
This is a procedure call
TYPE ← 0

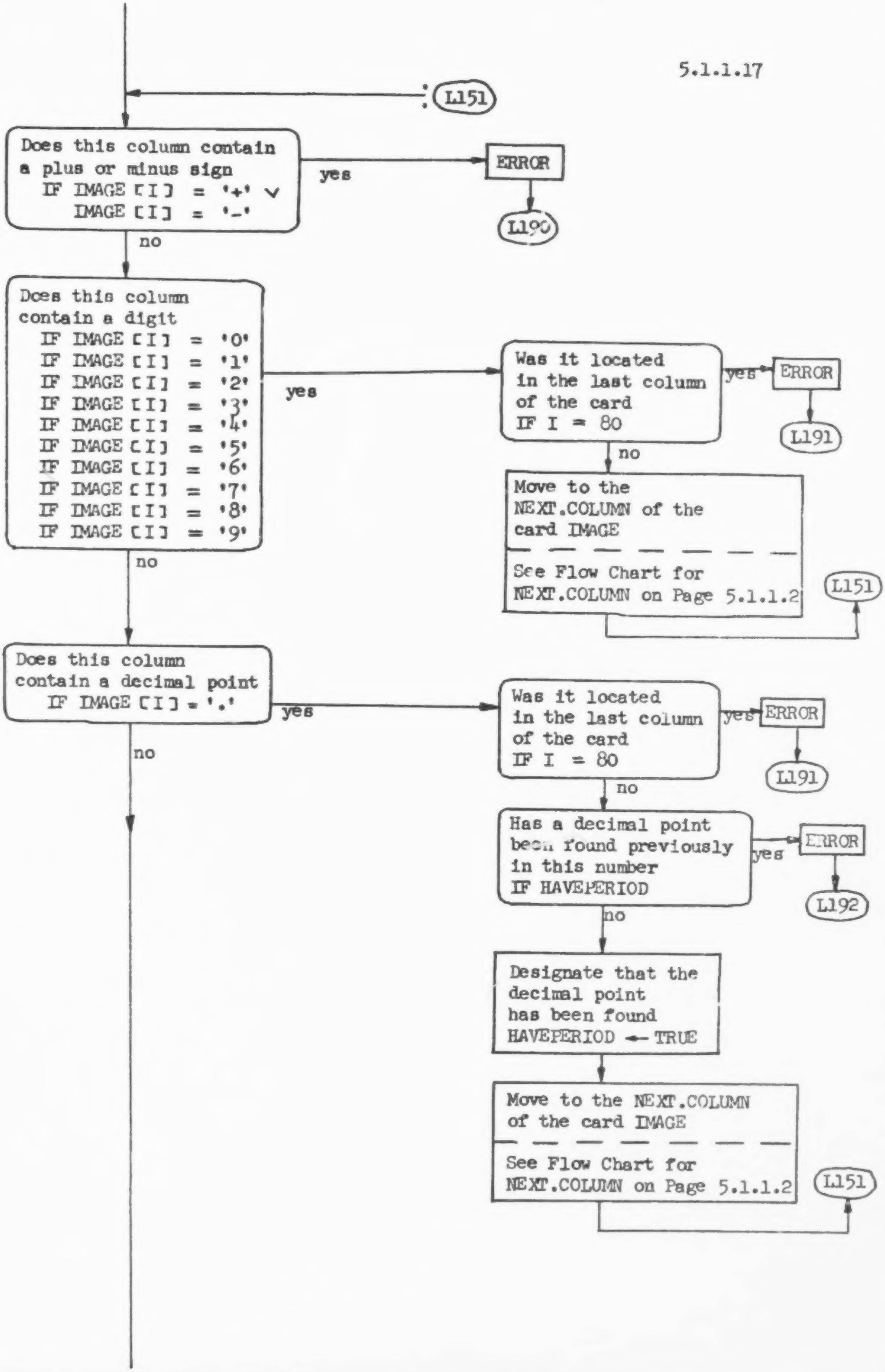

L115
Does this card contain a data variable IF TYPE = 1
yes
L130
no
Then it contains a procedure call
Recognize the first letter of the procedure call
IF IMAGE [I] = 'A'
IF IMAGE [I] = 'B'
etc.
Has a first letter been recognized
no
ERROR
L184
yes
Go to the appropriate procedure call label depending upon the letter recognized
THEN GO TO AP
THEN GO TO BP
etc.
Are there calls on PR at this label which have not been checked to try to recognize the procedure
no
Check if there are other segments of Phase 1 to be scanned in trying to recognize the procedure call
IF TRANSFER < SEGS
no
ERROR
L185
yes
L160
yes
Call PR for the next procedure name at that label
See Flow Chart for PR on Page 5.1.1.3
Was a match found between IMAG and the parameters of PR i.e. Was the procedure name recognized?
no
yes
Then an exit was made from PR to L110 or L199


L130
Recognize the first letter of the data variable
IF IMAGE [I] = 'A'
IF IMAGE [I] = 'B'
etc.
Has a first letter been recognized
no
ERROR
L186
yes
Go to the appropriate data variable label depending upon the letter recognized
THEN GO TO AD
THEN GO TO BD
etc.
Are there calls on SR or AR at this label which have not been checked to try to recognize the data variable
no
Check if there are other segments of Phase 1 to be scanned in trying to recognize the data variable
IF TRANSFER < SEGS
no
ERROR
L187
yes
L160
yes
Call SR or AR for the next data variable at this label
See Flow Chart for SR on Page 5.1.1.5
See Flow Chart for AR on Page 5.1.1.7
Was a match found between IMAG and the parameters of SR or AR i.e. Was the data variable recognized
no
yes
Then an exit was made from SR or AR to L150

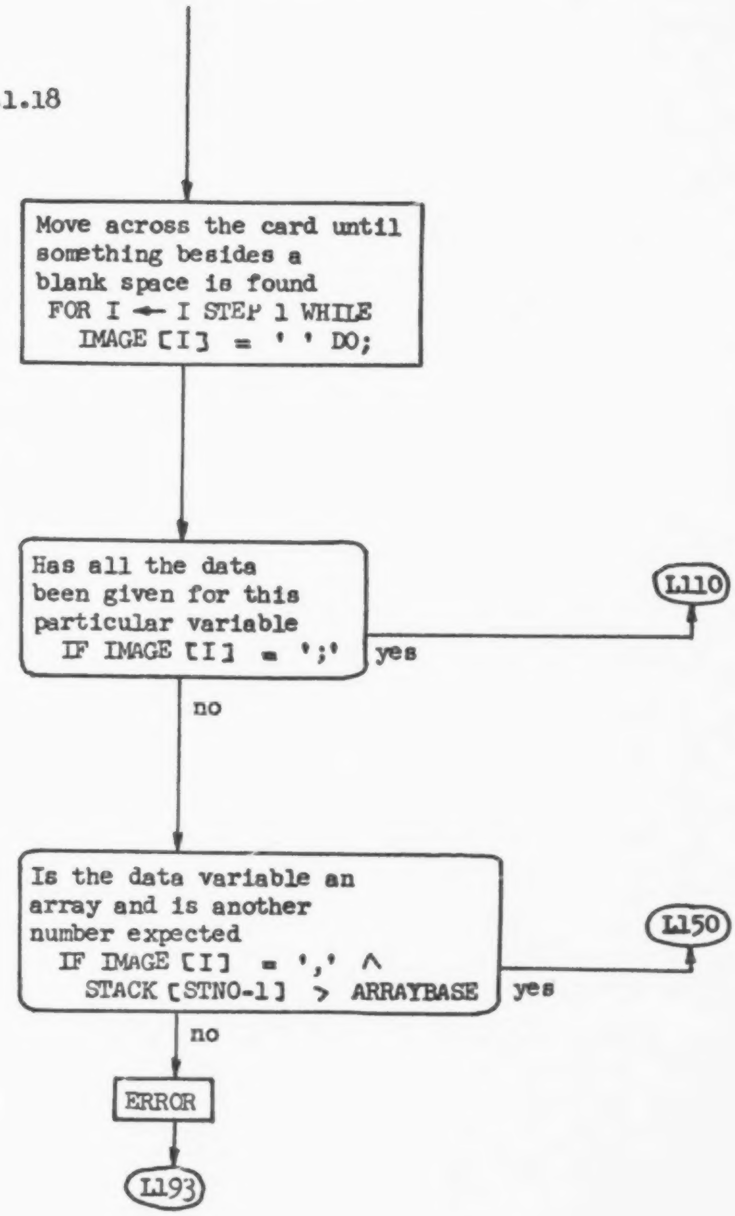

L150
Move to the NEXT.COLUMN of the card IMAGE
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2
Check if this data card has not been stored yet into DATABUFFER IF ¬ STORED
yes
no
Read the 20 four-character words of the data card into the next 20 words of DATABUFFER by means of NAME & READ statements
Fill the next word of DATABUFFER with a blank DATABUFFER [DANO+1] ← ' '
Increment the counter on the DATABUFFER by two DANO ← DANO+2
Set the Boolean switch to show that this card image has been read into DATABUFFER and will subsequently be read onto the end of the SCRATCH AREA STORED ← TRUE
Is the DATABUFFER filled so that it can be written onto the SCRATCH AREA DANO ≥ 1600
no
yes
Reset the pointer on DATABUFFER to one DANO ← 1
DISCWRITE the first 1600 words of DATABUFFER onto the SCRATCH AREA starting at DATABLOCK
Increment the DATABLOCK by 5 since 5 blocks of 320 words each have been written onto the SCRATCH AREA from DATABUFFER


Transfer the end of the last data card read into DATABUFFER to the beginning of DATABUFFER
FOR DANO ← DANO STEP 1 UNTIL EXCESS DO DATABUFFER [DANO] ← DATABUFFER [DANO+1600]
Reduce the EXCESS by four for the next time DATABUFFER fills beyond 1600
Has EXCESS become negative
IF EXCESS = -4
IF EXCESS = -3
IF EXCESS = -2
IF EXCESS = -1
no
yes
Reset EXCESS to a positive value again
EXCESS ← 17
EXCESS ← 18
EXCESS ← 19
EXCESS ← 20
Is this column of the data card blank
IMAGE [I] = ' '
yes
L150
no
Is there anything in this column besides the beginning of a legal piece of data
IF IMAGE [I] ≠ '+'
IF IMAGE [I] ≠ '-'
IF IMAGE [I] ≠ '0'
IF IMAGE [I] ≠ '1'
IF IMAGE [I] ≠ '2'
IF IMAGE [I] ≠ '3'
IF IMAGE [I] ≠ '4'
IF IMAGE [I] ≠ '5'
IF IMAGE [I] ≠ '6'
IF IMAGE [I] ≠ '7'
IF IMAGE [I] ≠ '8'
IF IMAGE [I] ≠ '9'
IF IMAGE [I] ≠ '.'
IF IMAGE [I] ≠ '*'
yes
ERROR
L188
no


Is there alphanumeric data on the card
IF IMAGE [I] = '*'
yes
L152
no
Then there is numeric data on the card
Had alphanumeric data previously been stored into this variable
IF TYPEDATA = 'ALPH'
yes
ERROR
L189
no
Designate the type of data as numeric
TYPEDATA ← 'NUMR'
Does this column contain a plus or minus sign before a number
IF IMAGE [I] = '+' ∨
IMAGE [I] = '-'
yes
Move to the NEXT.COLUMN of the card IMAGE
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2
no
Designate that a decimal point has not yet been found in this number
HAVEPERIOD ← FALSE


L151
Does this column contain a plus or minus sign
IF IMAGE [I] = '+' ∨ IMAGE [I] = '-'
yes
ERROR
L190
no
Does this column contain a digit
IF IMAGE [I] = '0'
IF IMAGE [I] = '1'
IF IMAGE [I] = '2'
IF IMAGE [I] = '3'
IF IMAGE [I] = '4'
IF IMAGE [I] = '5'
IF IMAGE [I] = '6'
IF IMAGE [I] = '7'
IF IMAGE [I] = '8'
IF IMAGE [I] = '9'
yes
Was it located in the last column of the card
IF I = 80
yes
ERROR
L191
no
Move to the NEXT.COLUMN of the card IMAGE
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2
L151
no
Does this column contain a decimal point
IF IMAGE [I] = '.'
yes
Was it located in the last column of the card
IF I = 80
yes
ERROR
L191
no
Has a decimal point been found previously in this number
IF HAVEPERIOD
yes
ERROR
L192
no
Designate that the decimal point has been found
HAVEPERIOD ← TRUE
Move to the NEXT.COLUMN of the card IMAGE
See Flow Chart for NEXT.COLUMN on Page 5.1.1.2
L151
no

5.1.1.18



Move across the card until something besides a blank space is found

```
FOR I ← I STEP 1 WHILE
  IMAGE [I] = ' ' DO;
```

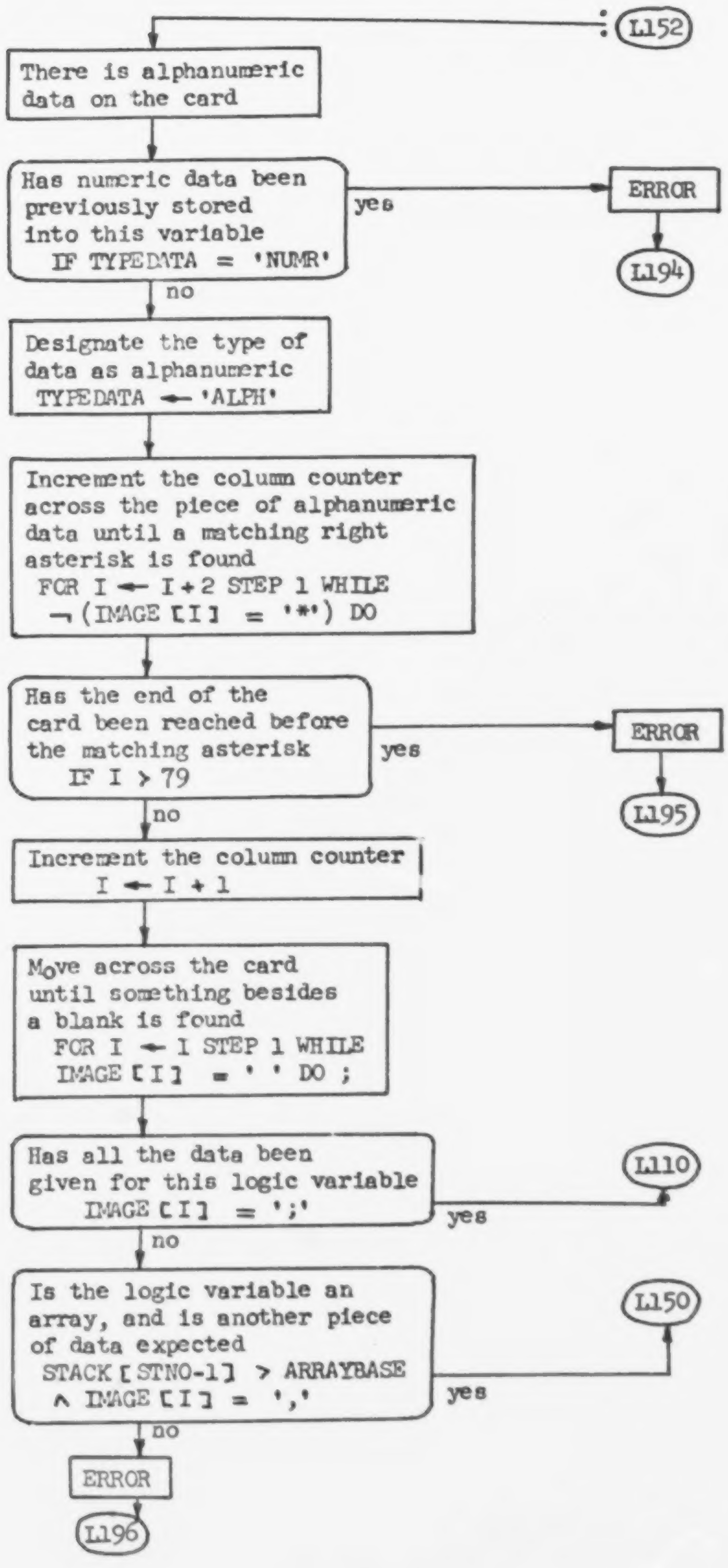
Has all the data been given for this particular variable

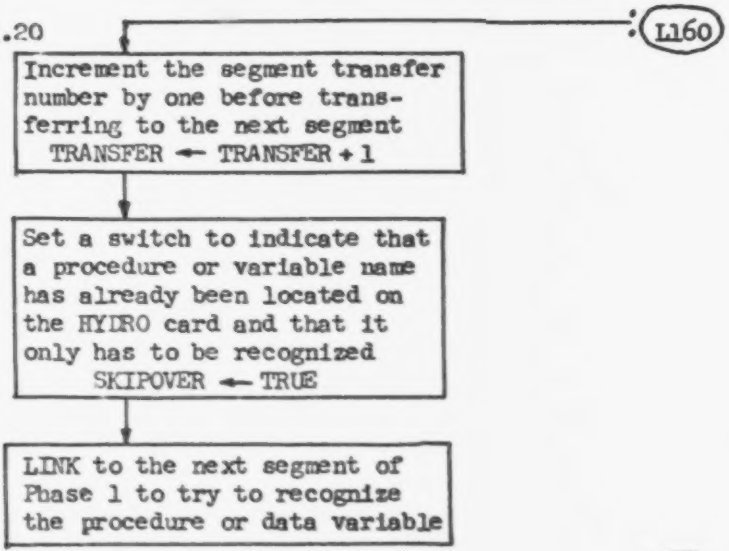
```
IF IMAGE [I] = ';'
```

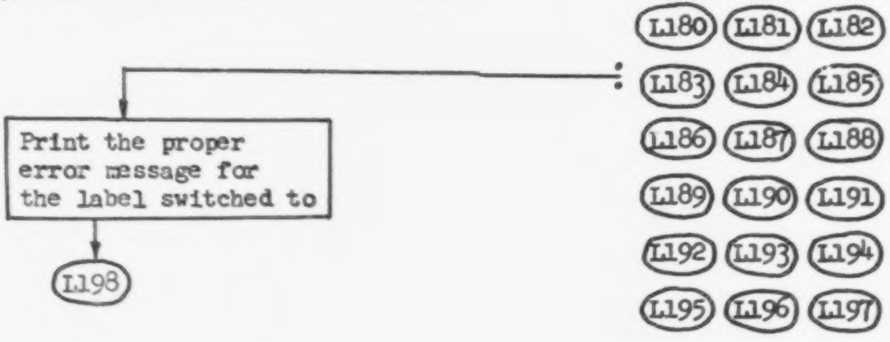
Is the data variable an array and is another number expected

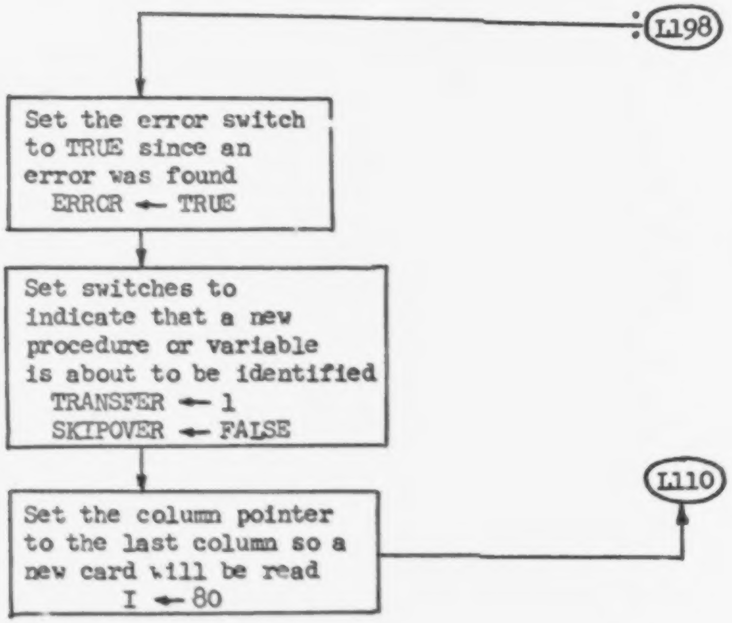
```
IF IMAGE [I] = ',' ∧
  STACK [STNO-1] > ARRAYBASE
```

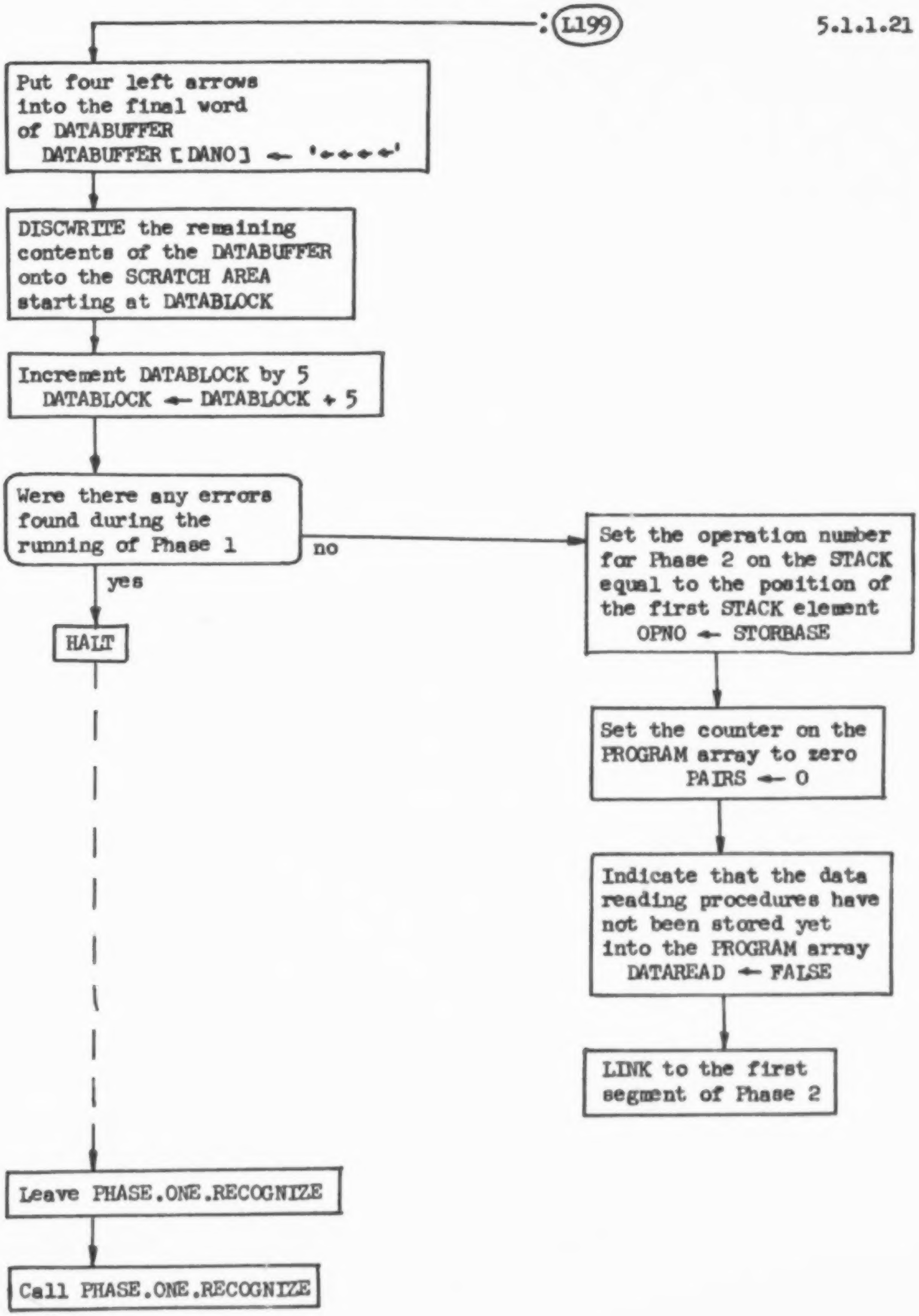
yes
L110
no
yes
L150
no
ERROR
L193


L152
There is alphanumeric data on the card
Has numeric data been previously stored into this variable
IF TYPEDATA = 'NUMR'
yes
ERROR
L194
no
Designate the type of data as alphanumeric
TYPEDATA ← 'ALPH'
Increment the column counter across the piece of alphanumeric data until a matching right asterisk is found
FOR I ← I + 2 STEP 1 WHILE
¬ (IMAGE [I] = '*') DO
Has the end of the card been reached before the matching asterisk
IF I > 79
yes
ERROR
L195
no
Increment the column counter
I ← I + 1
Move across the card until something besides a blank is found
FOR I ← I STEP 1 WHILE
IMAGE [I] = ' ' DO ;
Has all the data been given for this logic variable
IMAGE [I] = ';'
yes
L110
no
Is the logic variable an array, and is another piece of data expected
STACK [STNO-1] > ARRAYBASE
∧ IMAGE [I] = ','
yes
L150
no
ERROR
L196

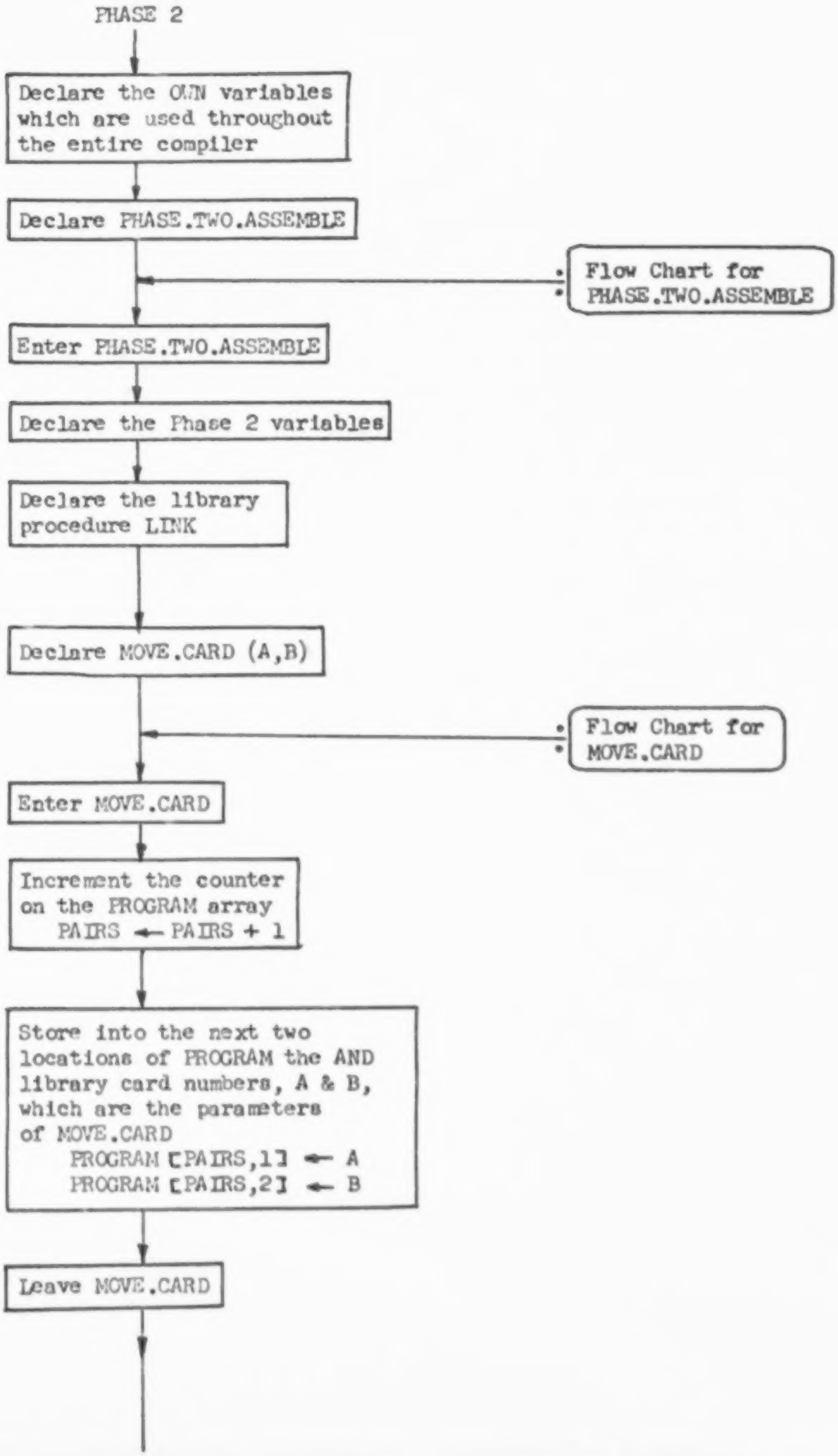

L160
Increment the segment transfer number by one before transferring to the next segment
TRANSFER ← TRANSFER + 1
Set a switch to indicate that a procedure or variable name has already been located on the HYDRO card and that it only has to be recognized
SKIPOVER ← TRUE
LINK to the next segment of Phase 1 to try to recognize the procedure or data variable


L180
L181
L182
L183
L184
L185
L186
L187
L188
L189
L190
L191
L192
L193
L194
L195
L196
L197
Print the proper error message for the label switched to
L198

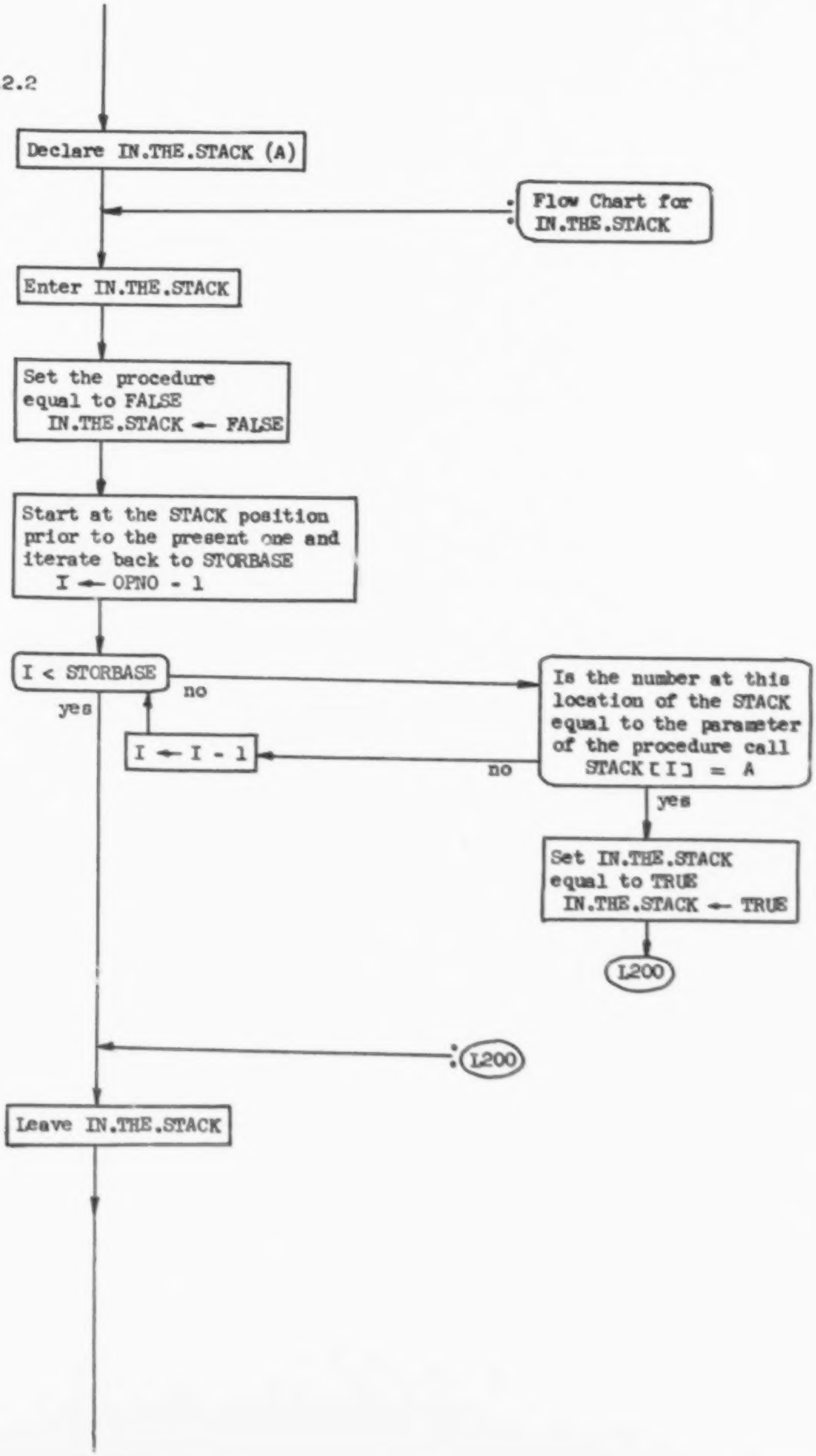

L198
Set the error switch to TRUE since an error was found
ERROR ← TRUE
Set switches to indicate that a new procedure or variable is about to be identified
TRANSFER ← 1
SKIPOVER ← FALSE
Set the column pointer to the last column so a new card will be read
I ← 80
L110

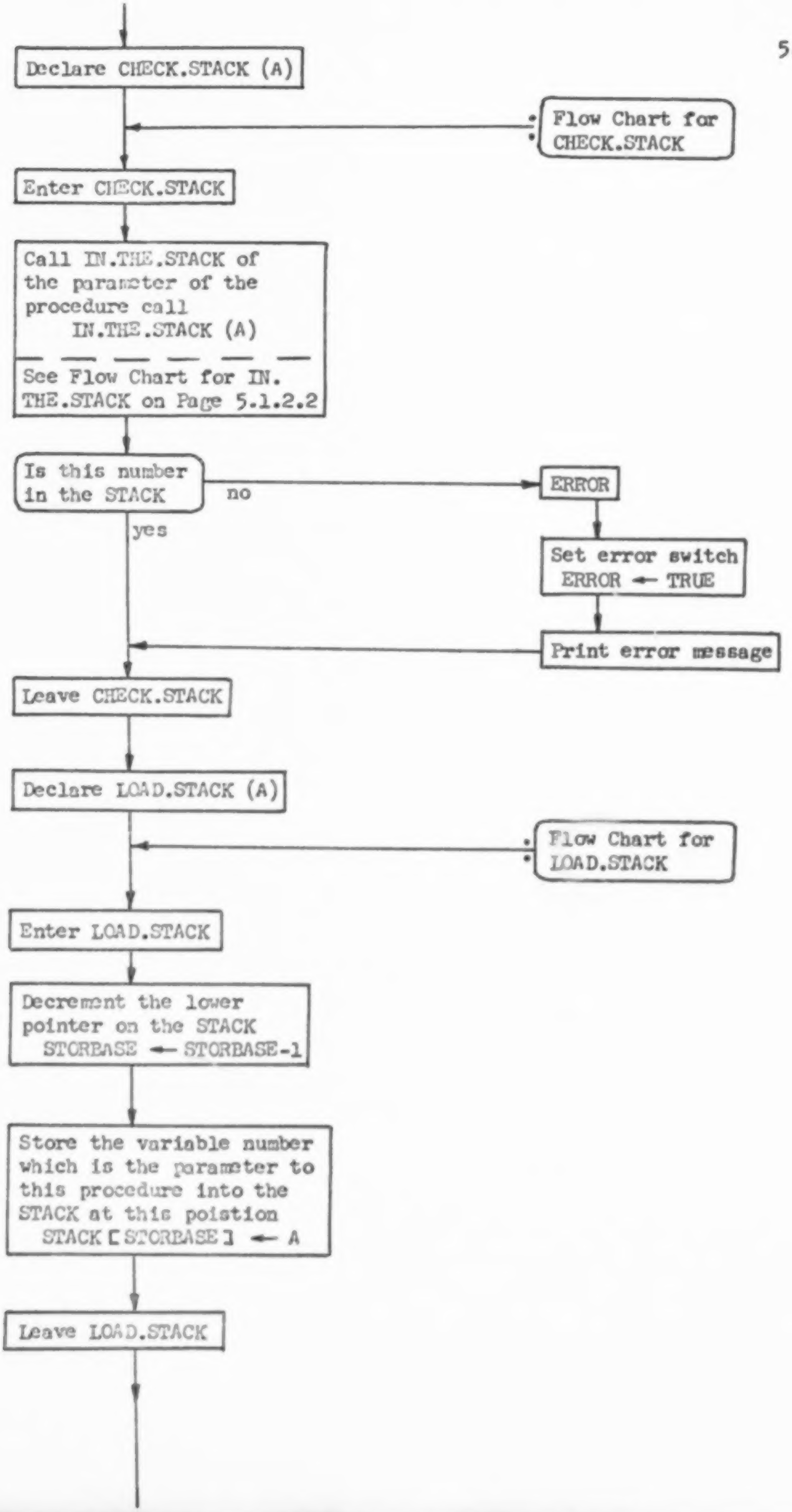

L199
Put four left arrows into the final word of DATABUFFER
DATABUFFER [DANO] ← '←←←←'
DISCWRITE the remaining contents of the DATABUFFER onto the SCRATCH AREA starting at DATABLOCK
Increment DATABLOCK by 5
DATABLOCK ← DATABLOCK + 5
Were there any errors found during the running of Phase 1
no
yes
HALT
Set the operation number for Phase 2 on the STACK equal to the position of the first STACK element
OPNO ← STORBASE
Set the counter on the PROGRAM array to zero
PAIRS ← 0
Indicate that the data reading procedures have not been stored yet into the PROGRAM array
DATAREAD ← FALSE
LINK to the first segment of Phase 2
Leave PHASE.ONE.RECOGNIZE
Call PHASE.ONE.RECOGNIZE

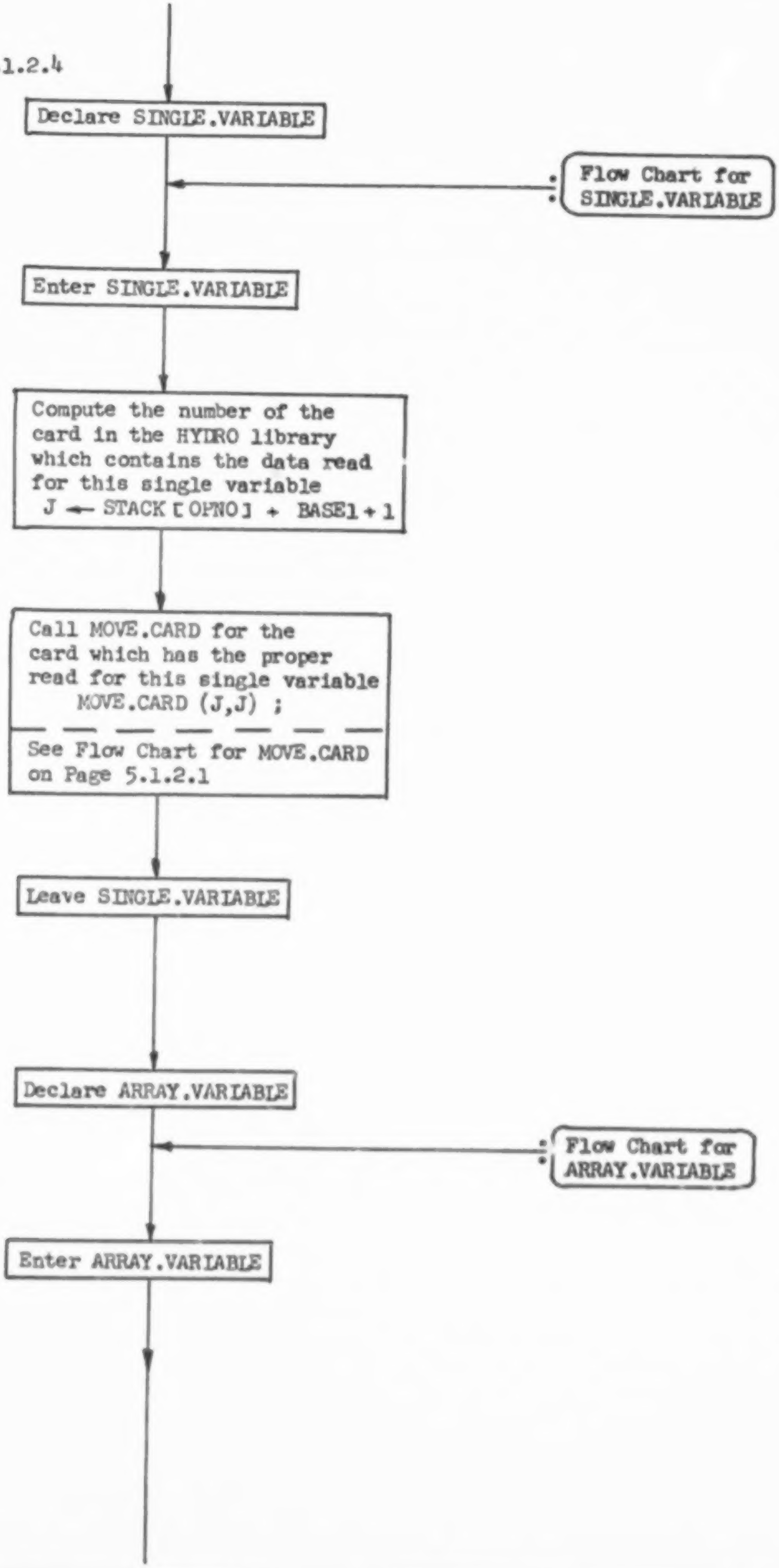
5.1.2 Phase Two - Assembly

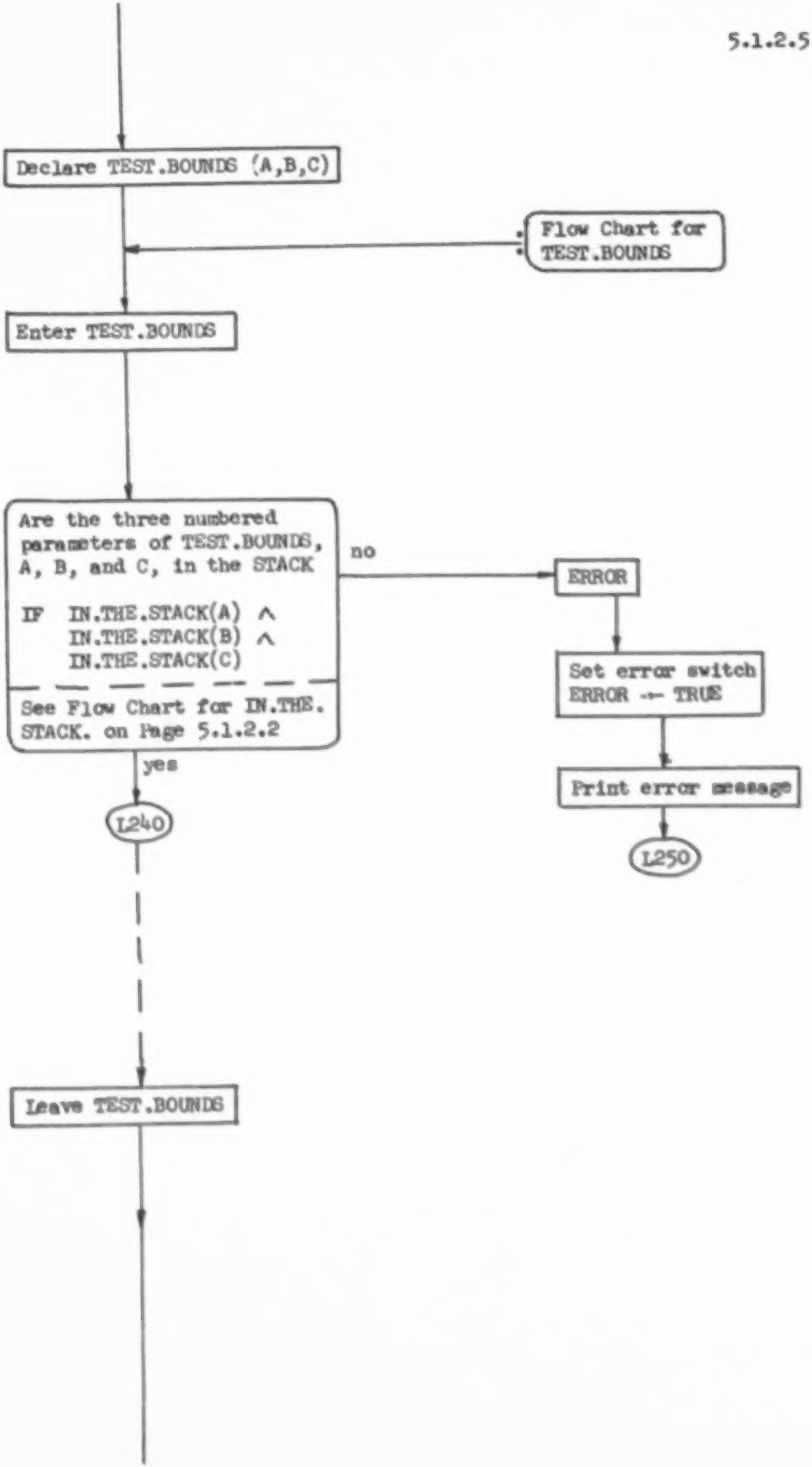

PHASE 2
Declare the OWN variables which are used throughout the entire compiler
Declare PHASE.TWO.ASSEMBLE
Flow Chart for PHASE.TWO.ASSEMBLE
Enter PHASE.TWO.ASSEMBLE
Declare the Phase 2 variables
Declare the library procedure LINK
Declare MOVE.CARD (A,B)
Flow Chart for MOVE.CARD
Enter MOVE.CARD
Increment the counter on the PROGRAM array
PAIRS ← PAIRS + 1
Store into the next two locations of PROGRAM the AND library card numbers, A & B, which are the parameters of MOVE.CARD
PROGRAM [PAIRS,1] ← A
PROGRAM [PAIRS,2] ← B
Leave MOVE.CARD

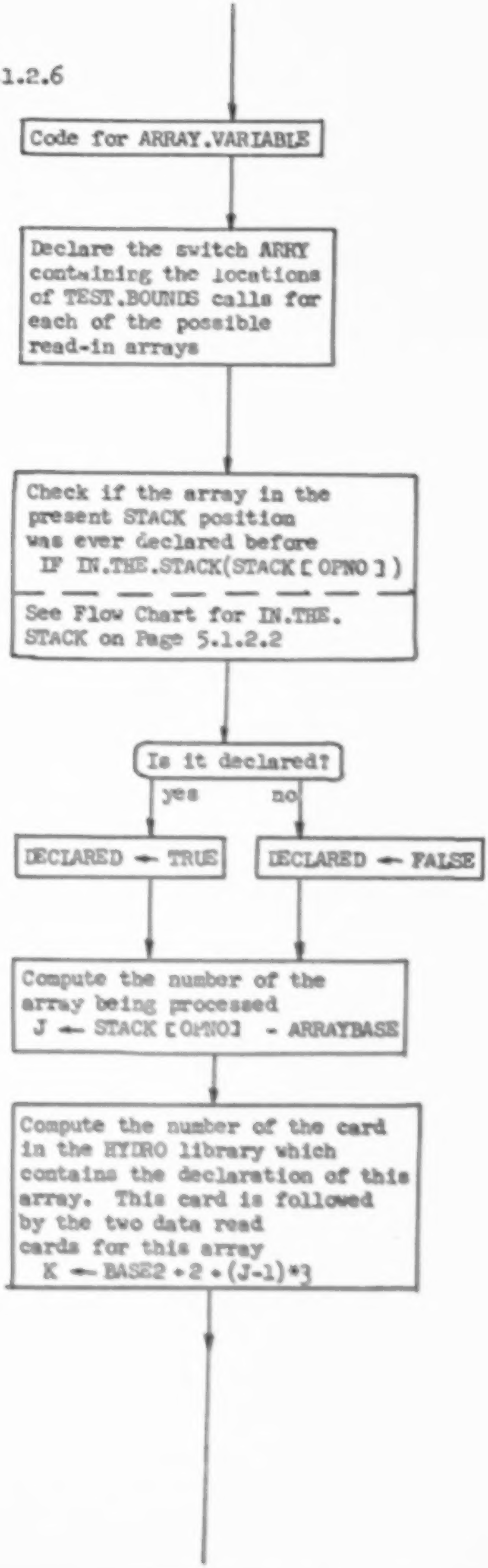
5.1.2.2

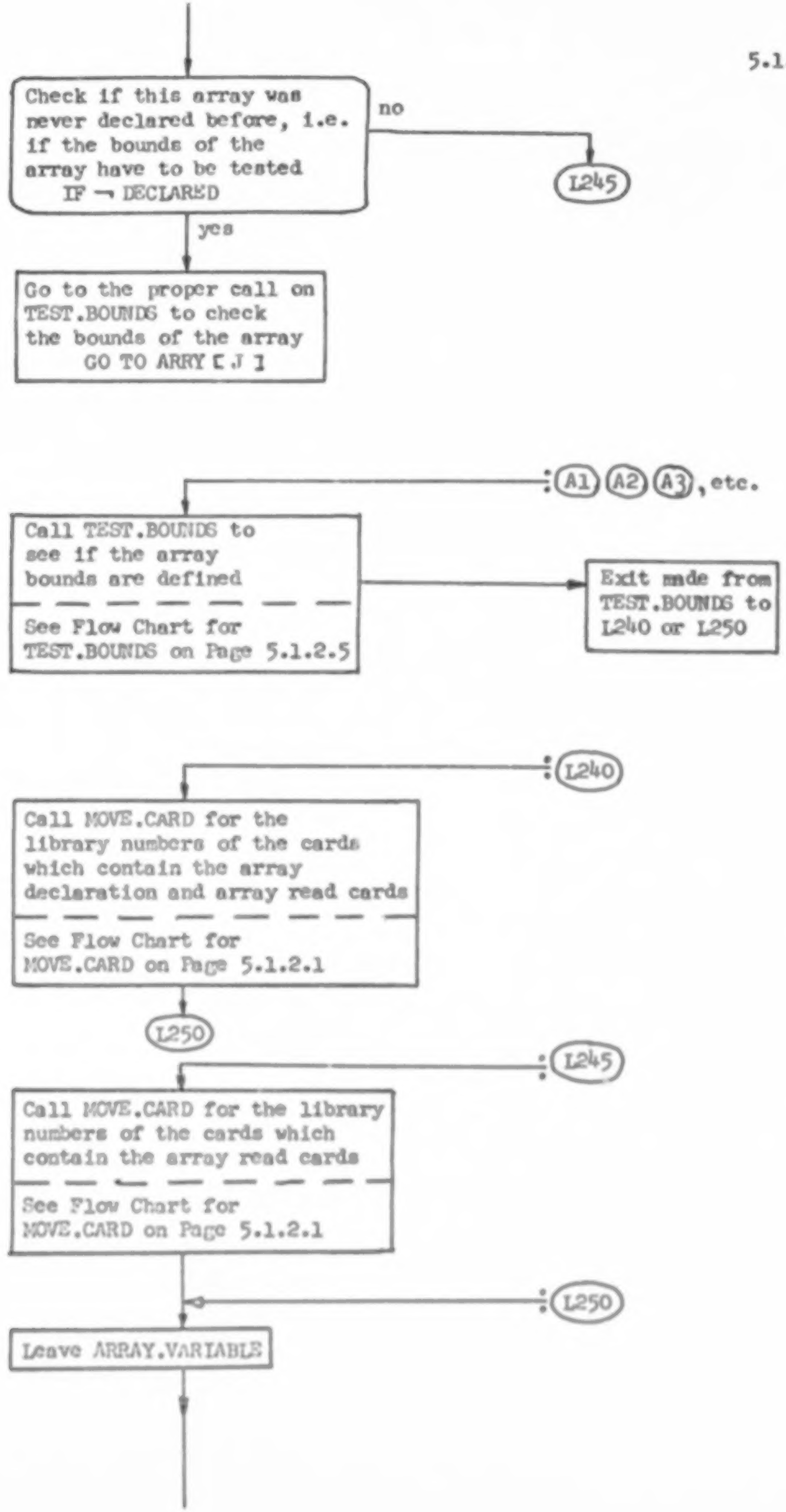

Declare IN.THE.STACK (A)
Flow Chart for IN.THE.STACK
Enter IN.THE.STACK
Set the procedure equal to FALSE
IN.THE.STACK ← FALSE
Start at the STACK position prior to the present one and iterate back to STORBASE
I ← OPNO - 1
I < STORBASE
no
yes
I ← I - 1
Is the number at this location of the STACK equal to the parameter of the procedure call
STACK [I] = A
no
yes
Set IN.THE.STACK equal to TRUE
IN.THE.STACK ← TRUE
L200
L200
Leave IN.THE.STACK

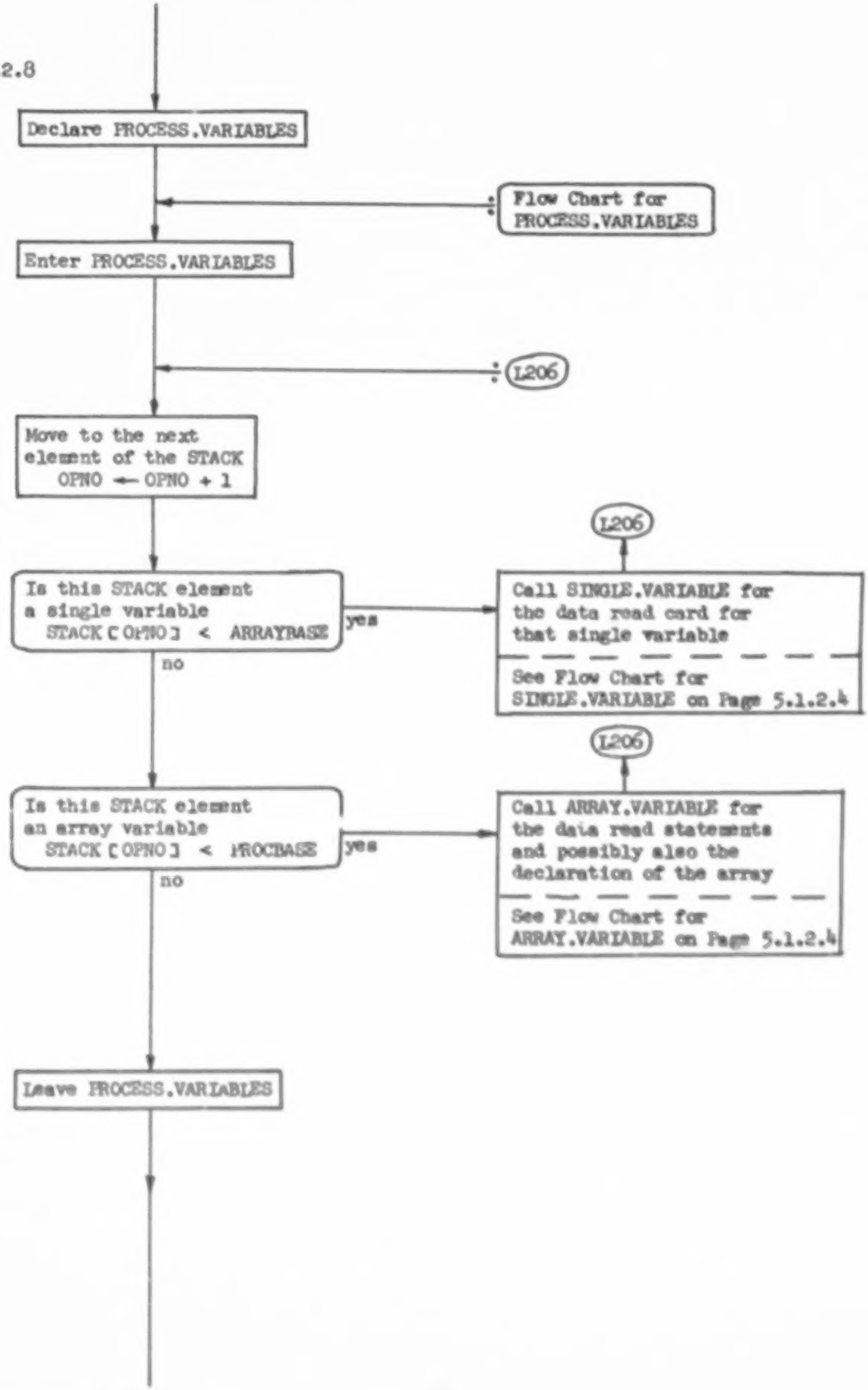

Declare CHECK.STACK (A)
Flow Chart for CHECK.STACK
Enter CHECK.STACK
Call IN.THE.STACK of the parameter of the procedure call IN.THE.STACK (A)
See Flow Chart for IN. THE.STACK on Page 5.1.2.2
Is this number in the STACK
no
yes
ERROR
Set error switch ERROR ← TRUE
Print error message
Leave CHECK.STACK
Declare LOAD.STACK (A)
Flow Chart for LOAD.STACK
Enter LOAD.STACK
Decrement the lower pointer on the STACK STORBASE ← STORBASE-1
Store the variable number which is the parameter to this procedure into the STACK at this poistion STACK [STORBASE] ← A
Leave LOAD.STACK

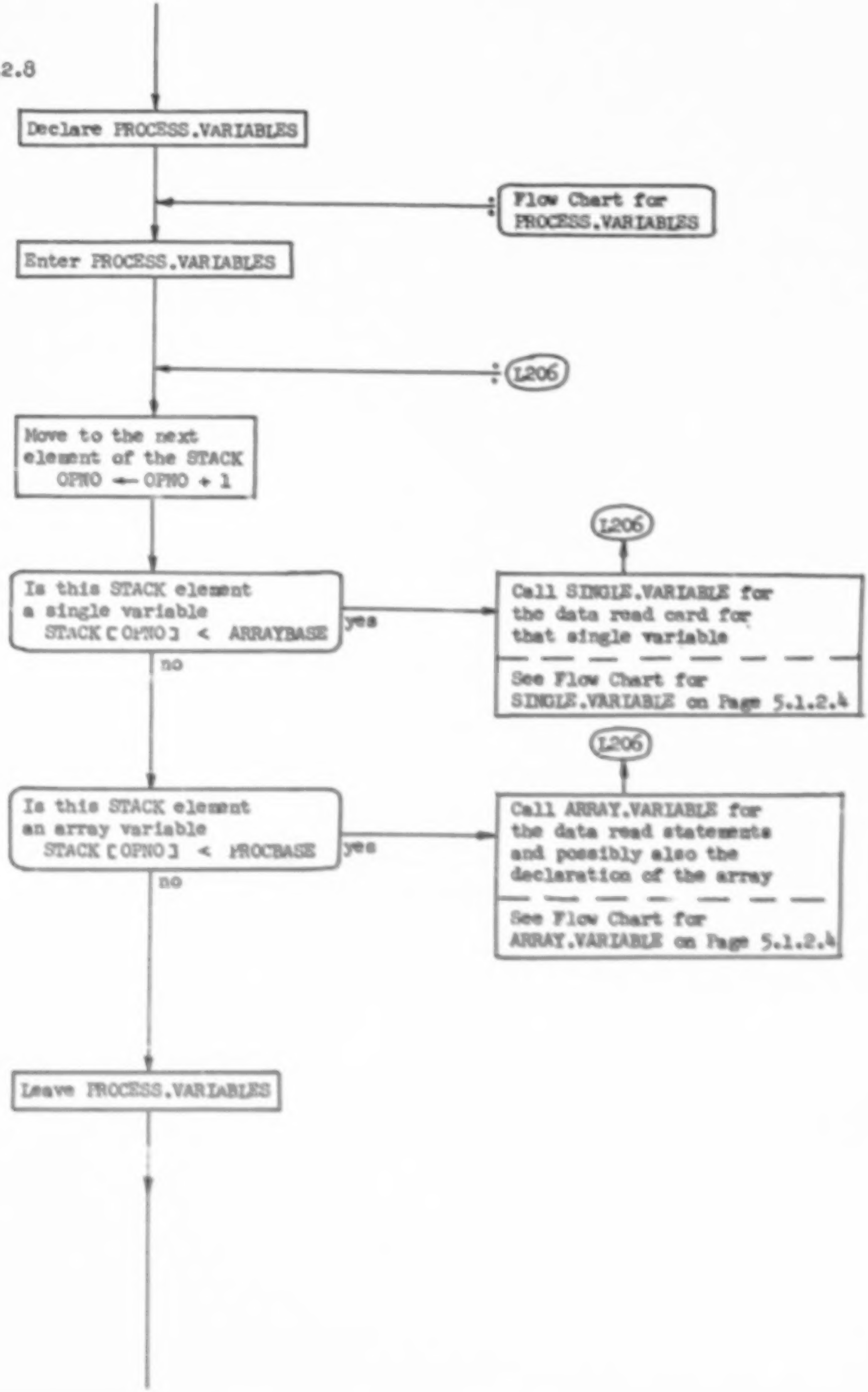

Declare SINGLE.VARIABLE
Flow Chart for SINGLE.VARIABLE
Enter SINGLE.VARIABLE
Compute the number of the card in the HYDRO library which contains the data read for this single variable
J ← STACK [OPNO] + BASE1 + 1
Call MOVE.CARD for the card which has the proper read for this single variable
MOVE.CARD (J,J) ;
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Leave SINGLE.VARIABLE
Declare ARRAY.VARIABLE
Flow Chart for ARRAY.VARIABLE
Enter ARRAY.VARIABLE

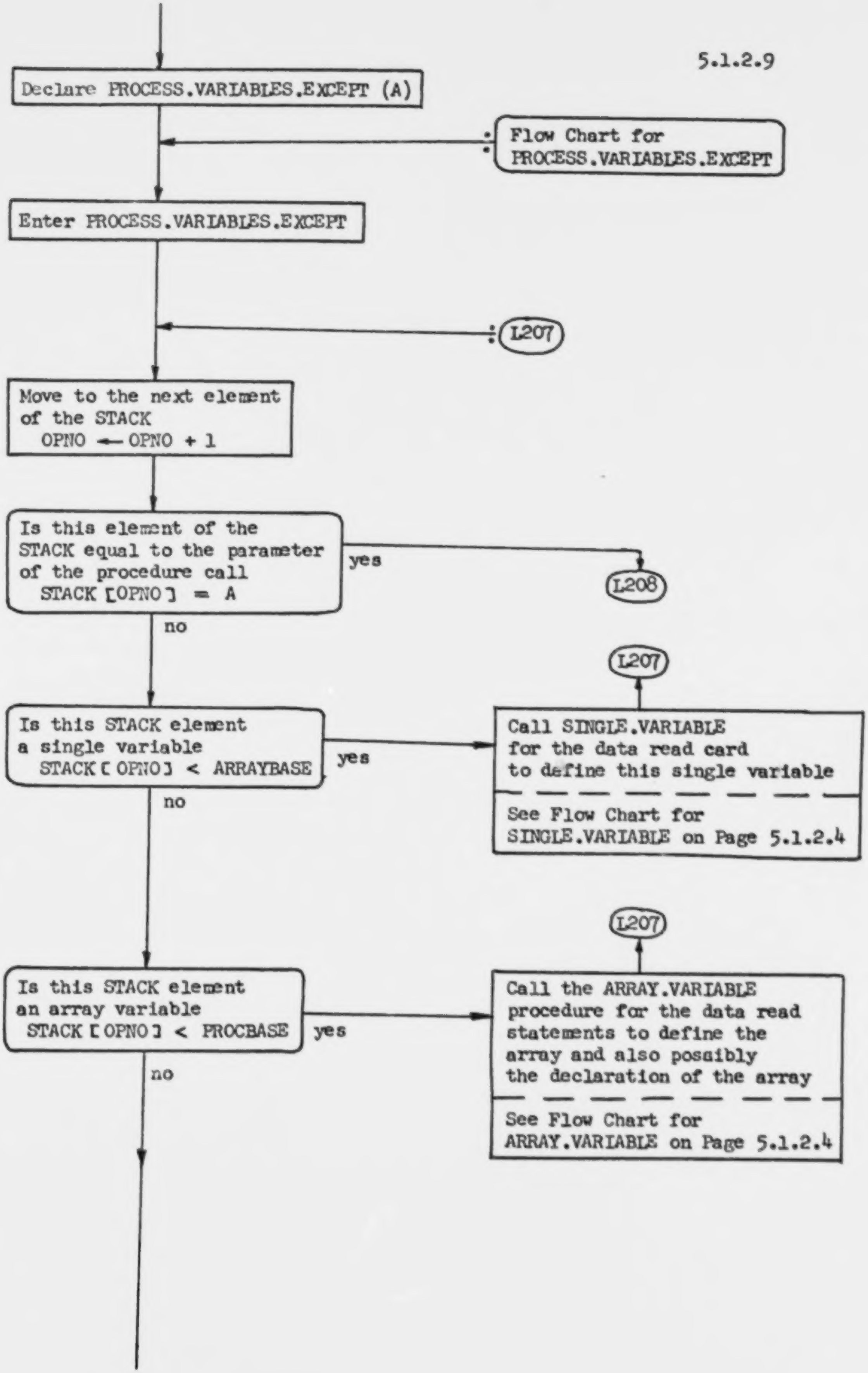

Declare TEST.BOUNDS (A,B,C)
Flow Chart for TEST.BOUNDS
Enter TEST.BOUNDS
Are the three numbered parameters of TEST.BOUNDS, A, B, and C, in the STACK
IF IN.THE.STACK(A) ∧ IN.THE.STACK(B) ∧ IN.THE.STACK(C)
See Flow Chart for IN.THE. STACK. on Page 5.1.2.2
no
ERROR
Set error switch ERROR ← TRUE
Print error message
L250
yes
L240
Leave TEST.BOUNDS


Code for ARRAY.VARIABLE
Declare the switch ARRY containing the locations of TEST.BOUNDS calls for each of the possible read-in arrays
Check if the array in the present STACK position was ever declared before
IF IN.THE.STACK(STACK [OPNO])
See Flow Chart for IN.THE. STACK on Page 5.1.2.2
Is it declared?
yes
no
DECLARED ← TRUE
DECLARED ← FALSE
Compute the number of the array being processed
J ← STACK [OPNO] - ARRAYBASE
Compute the number of the card in the HYDRO library which contains the declaration of this array. This card is followed by the two data read cards for this array
K ← BASE2 + 2 + (J-1)*3

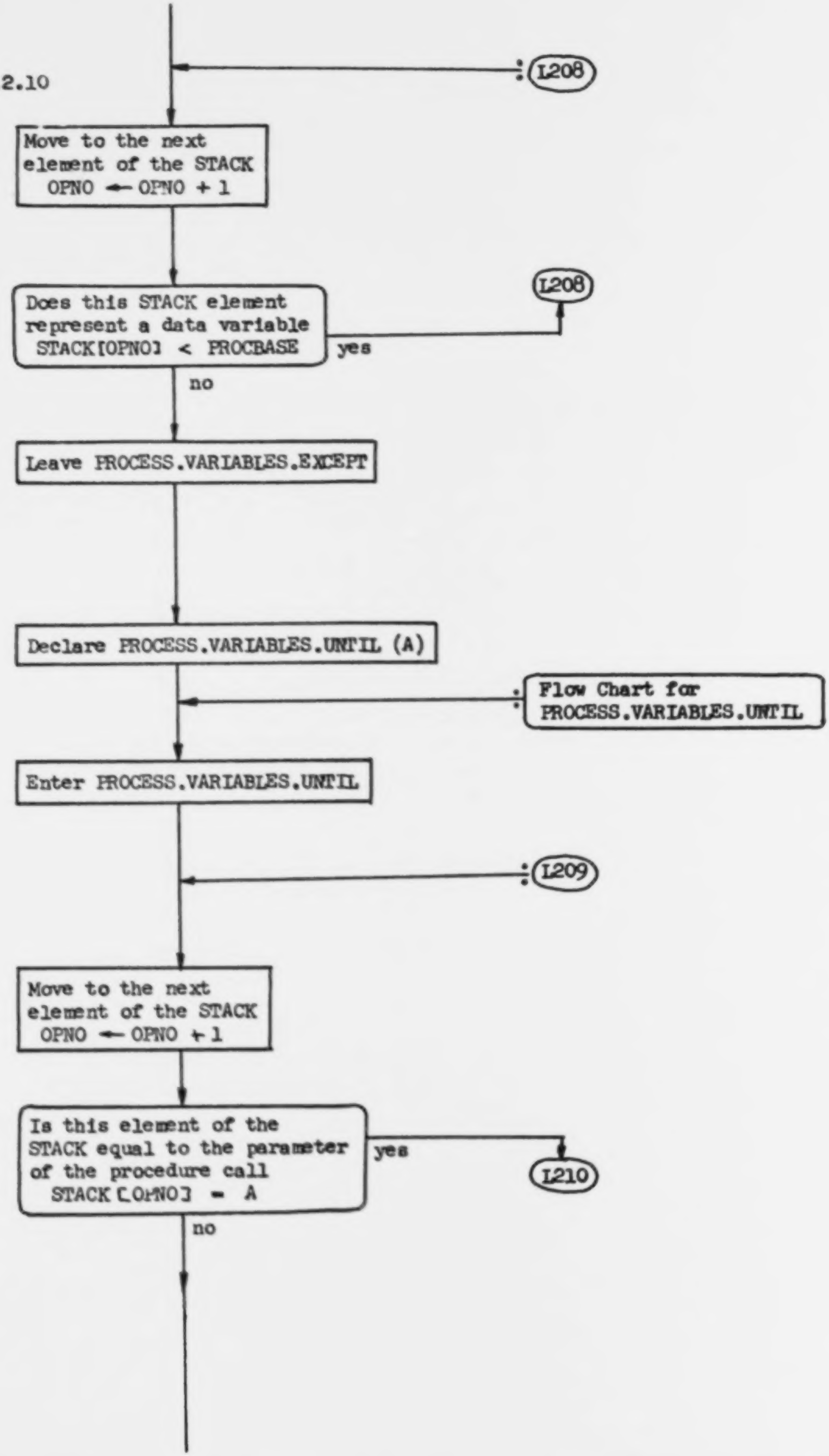

Check if this array was never declared before, i.e. if the bounds of the array have to be tested
IF ¬ DECLARED
no
L245
yes
Go to the proper call on TEST.BOUNDS to check the bounds of the array
GO TO ARRY [J]
A1
A2
A3
, etc.
Call TEST.BOUNDS to see if the array bounds are defined
See Flow Chart for TEST.BOUNDS on Page 5.1.2.5
Exit made from TEST.BOUNDS to L240 or L250
L240
Call MOVE.CARD for the library numbers of the cards which contain the array declaration and array read cards
See Flow Chart for MOVE.CARD on Page 5.1.2.1
L250
L245
Call MOVE.CARD for the library numbers of the cards which contain the array read cards
See Flow Chart for MOVE.CARD on Page 5.1.2.1
L250
Leave ARRAY.VARIABLE

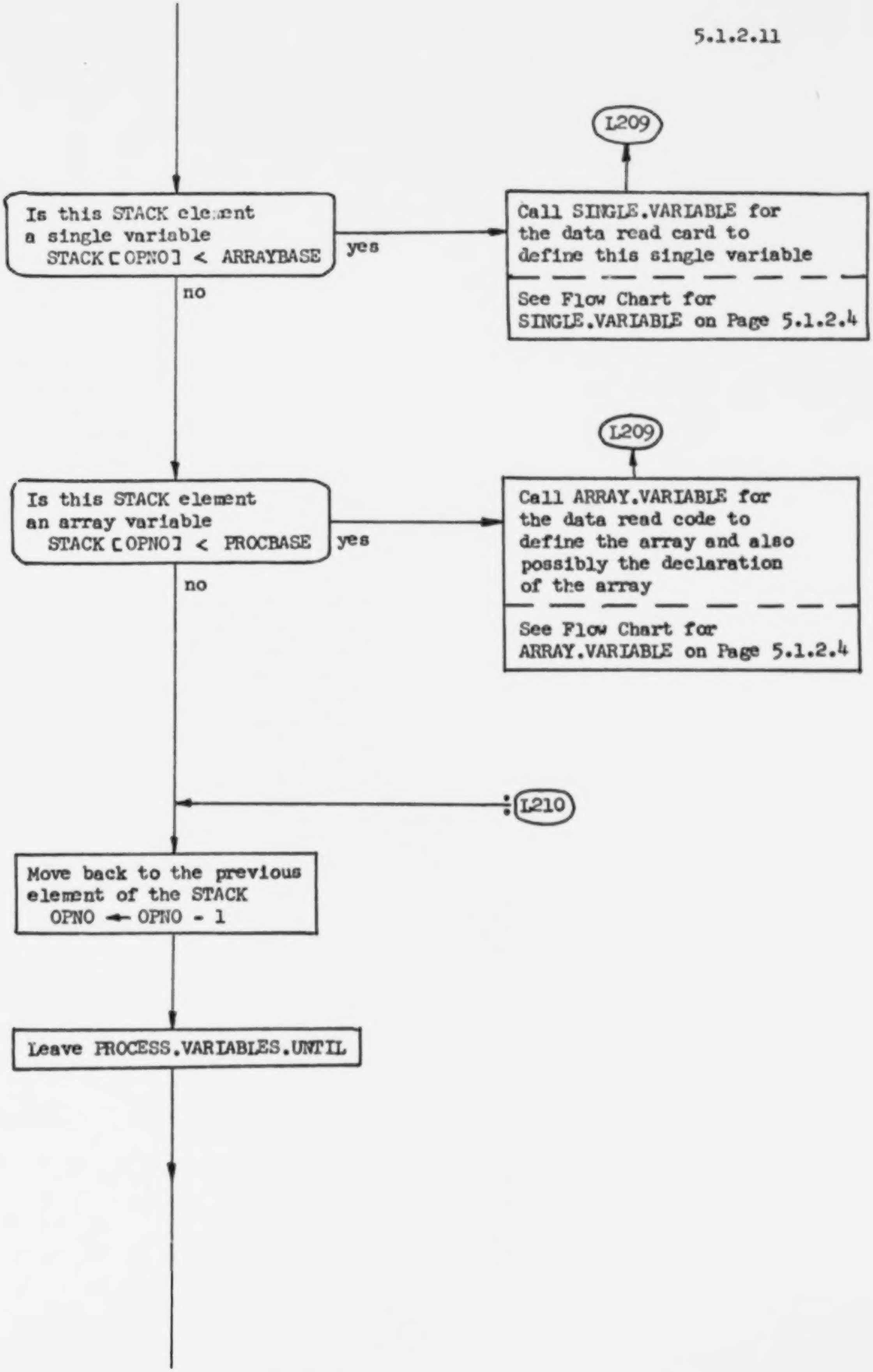

Declare PROCESS.VARIABLES
Flow Chart for PROCESS.VARIABLES
Enter PROCESS.VARIABLES
L206
Move to the next element of the STACK OPNO ← OPNO + 1
L206
Is this STACK element a single variable STACK [OPNO] < ARRAYBASE
yes
Call SINGLE.VARIABLE for the data read card for that single variable
See Flow Chart for SINGLE.VARIABLE on Page 5.1.2.4
no
L206
Is this STACK element an array variable STACK [OPNO] < PROCBASE
yes
Call ARRAY.VARIABLE for the data read statements and possibly also the declaration of the array
See Flow Chart for ARRAY.VARIABLE on Page 5.1.2.4
no
Leave PROCESS.VARIABLES

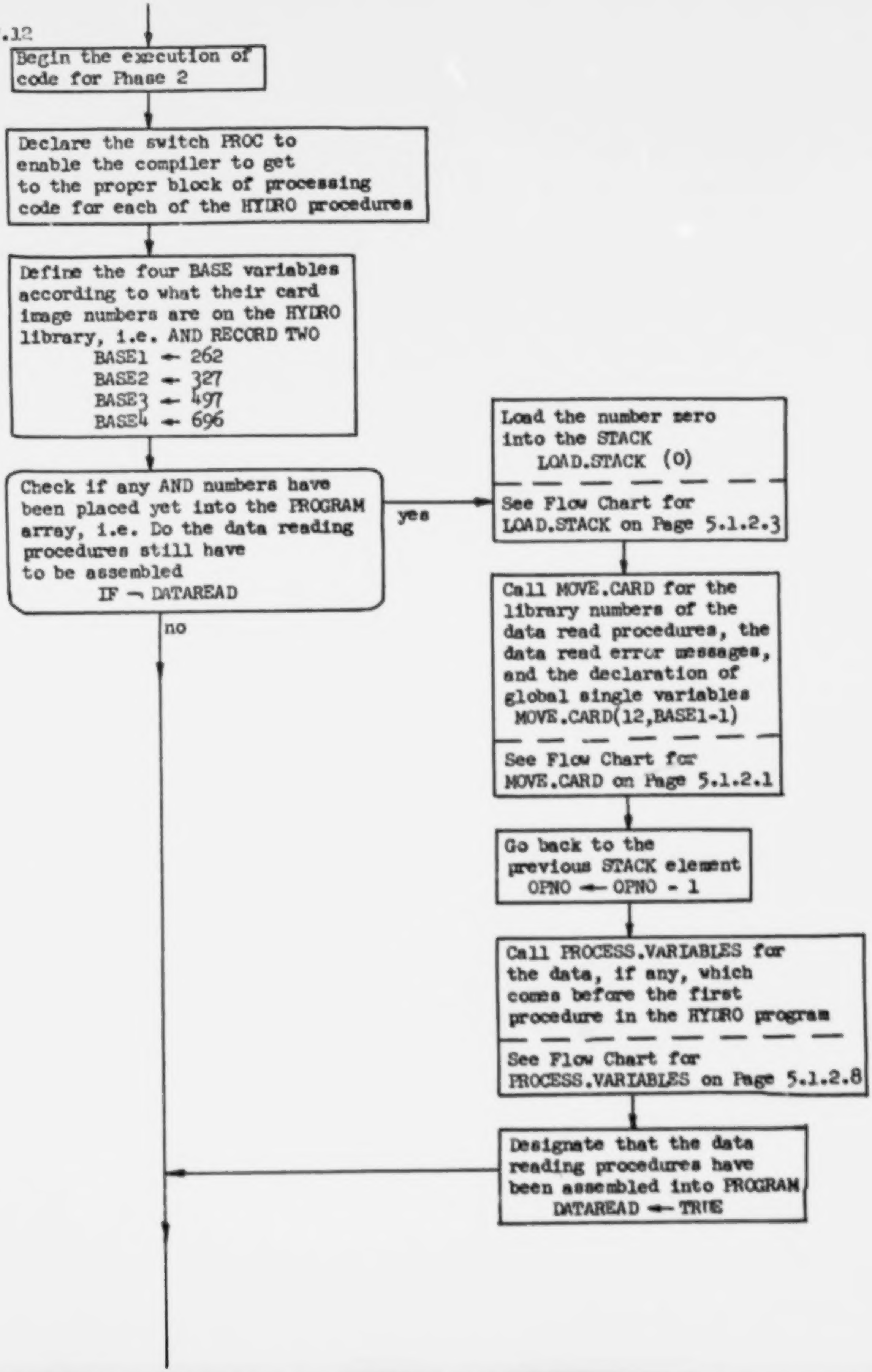

Declare PROCESS.VARIABLES
Flow Chart for PROCESS.VARIABLES
Enter PROCESS.VARIABLES
L206
Move to the next element of the STACK
OPNO ← OPNO + 1
L206
Is this STACK element a single variable
STACK [OPNO] < ARRAYBASE
yes
Call SINGLE.VARIABLE for the data read card for that single variable
See Flow Chart for SINGLE.VARIABLE on Page 5.1.2.4
no
L206
Is this STACK element an array variable
STACK [OPNO] < PROCBASE
yes
Call ARRAY.VARIABLE for the data read statements and possibly also the declaration of the array
See Flow Chart for ARRAY.VARIABLE on Page 5.1.2.4
no
Leave PROCESS.VARIABLES

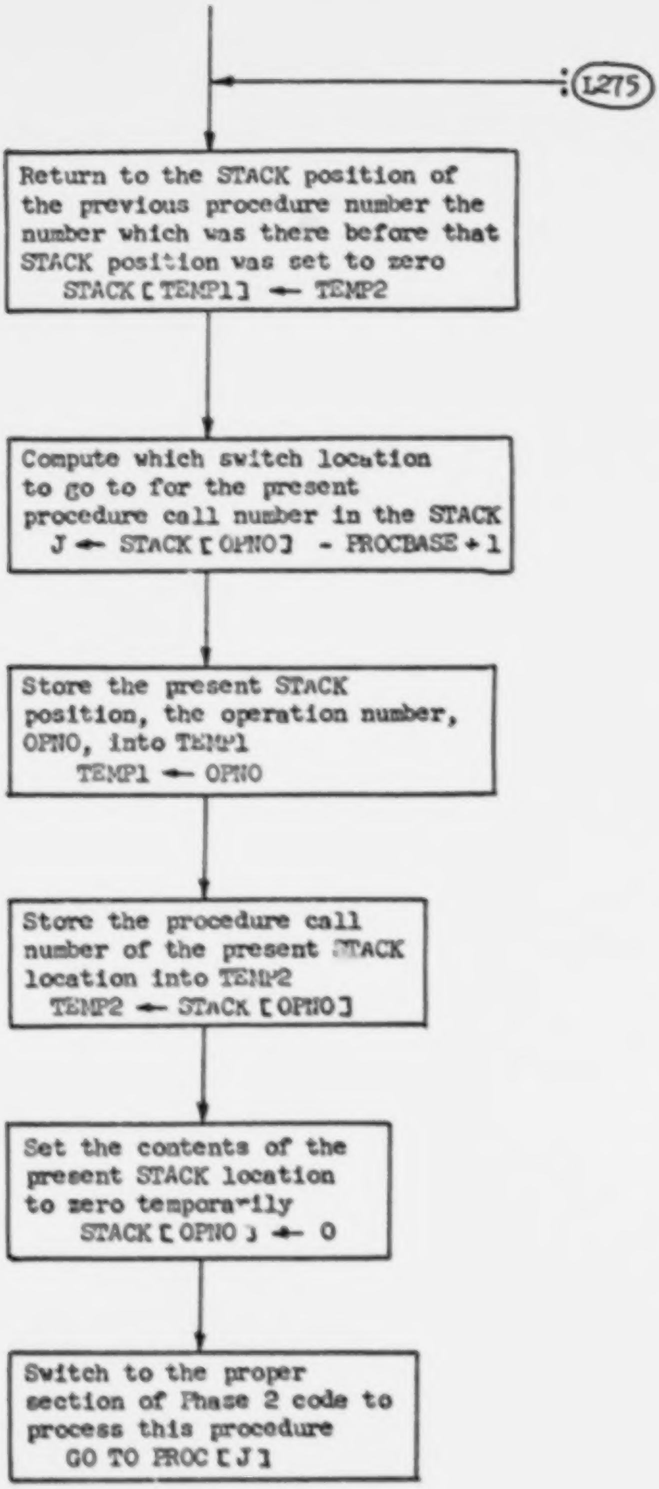

Declare PROCESS.VARIABLES.EXCEPT (A)
Flow Chart for PROCESS.VARIABLES.EXCEPT
Enter PROCESS.VARIABLES.EXCEPT
L207
Move to the next element of the STACK OPNO ← OPNO + 1
Is this element of the STACK equal to the parameter of the procedure call STACK [OPNO] = A
yes
L208
no
L207
Is this STACK element a single variable STACK [OPNO] < ARRAYBASE
yes
Call SINGLE.VARIABLE for the data read card to define this single variable
See Flow Chart for SINGLE.VARIABLE on Page 5.1.2.4
no
L207
Is this STACK element an array variable STACK [OPNO] < PROCBASE
yes
Call the ARRAY.VARIABLE procedure for the data read statements to define the array and also possibly the declaration of the array
See Flow Chart for ARRAY.VARIABLE on Page 5.1.2.4
no

5.1.2.10


L208
Move to the next element of the STACK OPNO ← OPNO + 1
Does this STACK element represent a data variable STACK[OPNO] < PROCBASE
yes
L208
no
Leave PROCESS.VARIABLES.EXCEPT
Declare PROCESS.VARIABLES.UNTIL (A)
Flow Chart for PROCESS.VARIABLES.UNTIL
Enter PROCESS.VARIABLES.UNTIL
L209
Move to the next element of the STACK OPNO ← OPNO + 1
Is this element of the STACK equal to the parameter of the procedure call STACK[OPNO] = A
yes
L210
no


L209
Is this STACK element a single variable STACK [OPNO] < ARRAYBASE
yes
no
Call SINGLE.VARIABLE for the data read card to define this single variable
See Flow Chart for SINGLE.VARIABLE on Page 5.1.2.4
L209
Is this STACK element an array variable STACK [OPNO] < PROCBASE
yes
no
Call ARRAY.VARIABLE for the data read code to define the array and also possibly the declaration of the array
See Flow Chart for ARRAY.VARIABLE on Page 5.1.2.4
L210
Move back to the previous element of the STACK OPNO ← OPNO - 1
Leave PROCESS.VARIABLES.UNTIL


Begin the execution of code for Phase 2
Declare the switch PROC to enable the compiler to get to the proper block of processing code for each of the HYDRO procedures
Define the four BASE variables according to what their card image numbers are on the HYDRO library, i.e. AND RECORD TWO
BASE1 ← 262
BASE2 ← 327
BASE3 ← 497
BASE4 ← 696
Check if any AND numbers have been placed yet into the PROGRAM array, i.e. Do the data reading procedures still have to be assembled
IF ¬ DATAREAD
yes
no
Load the number zero into the STACK
LOAD.STACK (0)
See Flow Chart for LOAD.STACK on Page 5.1.2.3
Call MOVE.CARD for the library numbers of the data read procedures, the data read error messages, and the declaration of global single variables
MOVE.CARD(12,BASE1-1)
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Go back to the previous STACK element
OPNO ← OPNO - 1
Call PROCESS.VARIABLES for the data, if any, which comes before the first procedure in the HYDRO program
See Flow Chart for PROCESS.VARIABLES on Page 5.1.2.8
Designate that the data reading procedures have been assembled into PROGRAM
DATAREAD ← TRUE



Return to the STACK position of the previous procedure number the number which was there before that STACK position was set to zero
STACK [TEMP1] ← TEMP2

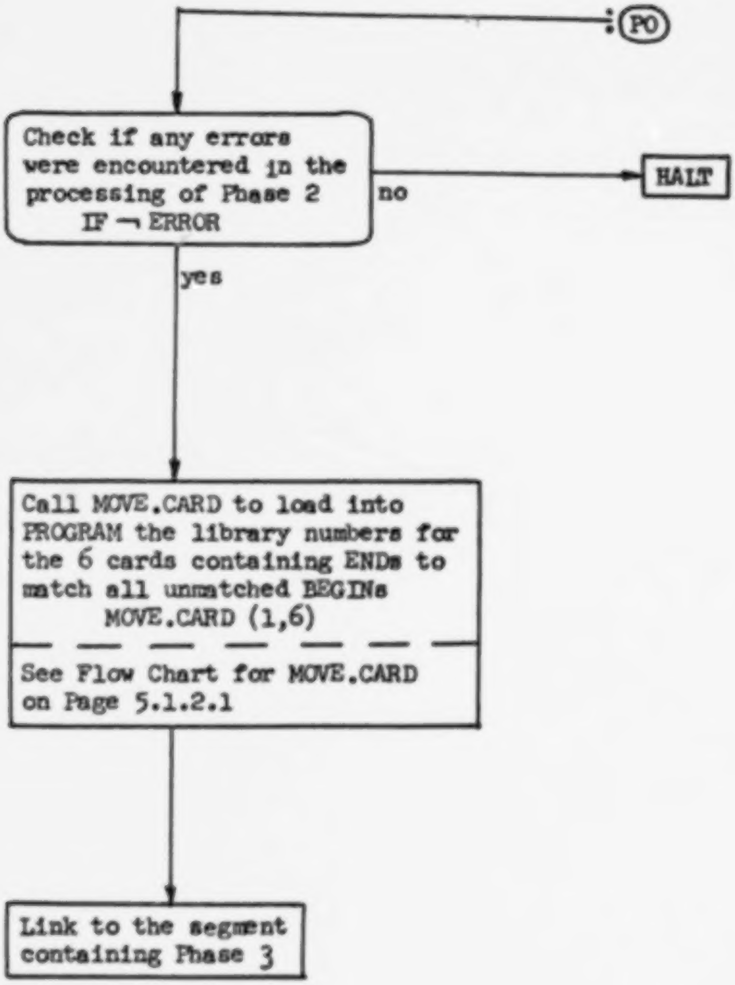
Compute which switch location to go to for the present procedure call number in the STACK
J ← STACK [OPNO] - PROCBASE + 1

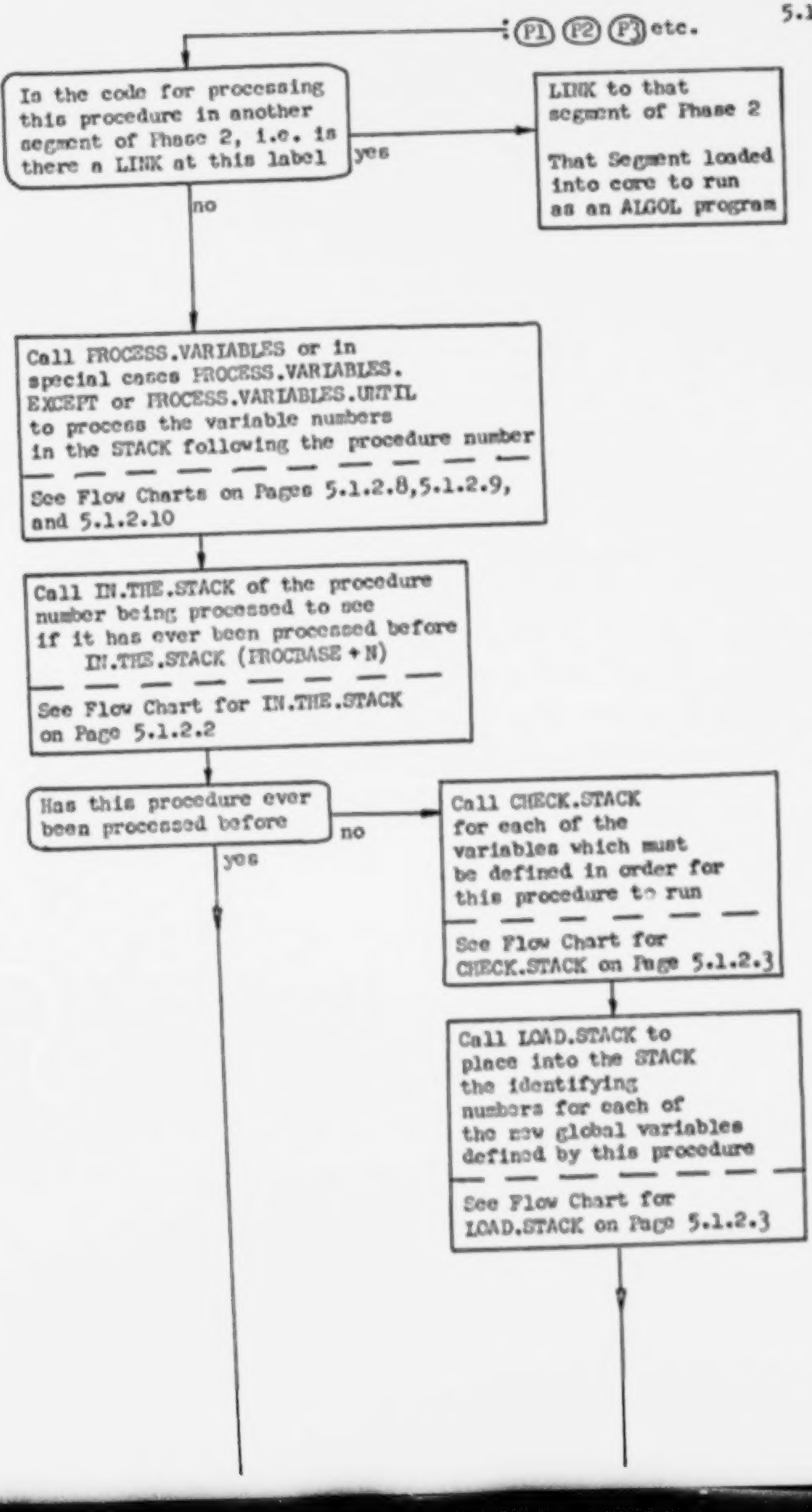
Store the present STACK position, the operation number, OPNO, into TEMP1
TEMP1 ← OPNO

Store the procedure call number of the present STACK location into TEMP2
TEMP2 ← STACK [OPNO]

Set the contents of the present STACK location to zero temporarily
STACK [OPNO] ← 0

Switch to the proper section of Phase 2 code to process this procedure
GO TO PROC [J]


PO
Check if any errors were encountered in the processing of Phase 2
IF ¬ ERROR
no
HALT
yes
Call MOVE.CARD to load into PROGRAM the library numbers for the 6 cards containing ENDs to match all unmatched BEGINs
MOVE.CARD (1,6)
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Link to the segment containing Phase 3



In the code for processing this procedure in another segment of Phase 2, i.e. is there a LINK at this label

LINK to that segment of Phase 2

That Segment loaded into core to run as an ALGOL program

Call PROCESS.VARIABLES or in special cases PROCESS.VARIABLES.EXCEPT or PROCESS.VARIABLES.UNTIL to process the variable numbers in the STACK following the procedure number

See Flow Charts on Pages 5.1.2.8, 5.1.2.9, and 5.1.2.10

Call IN.THE.STACK of the procedure number being processed to see if it has ever been processed before
IN.THE.STACK (PROCBASE + N)

See Flow Chart for IN.THE.STACK on Page 5.1.2.2

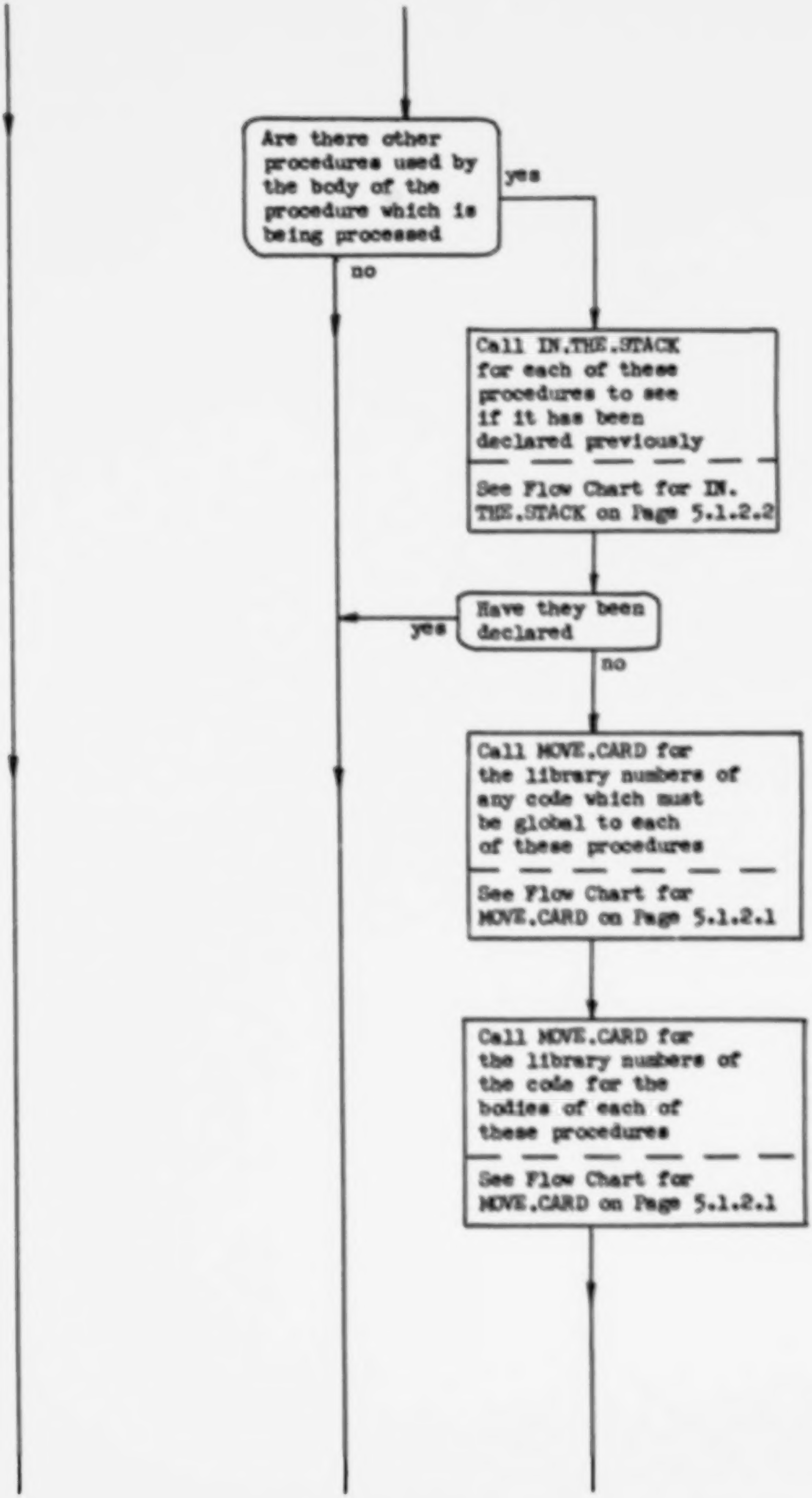
Has this procedure ever been processed before

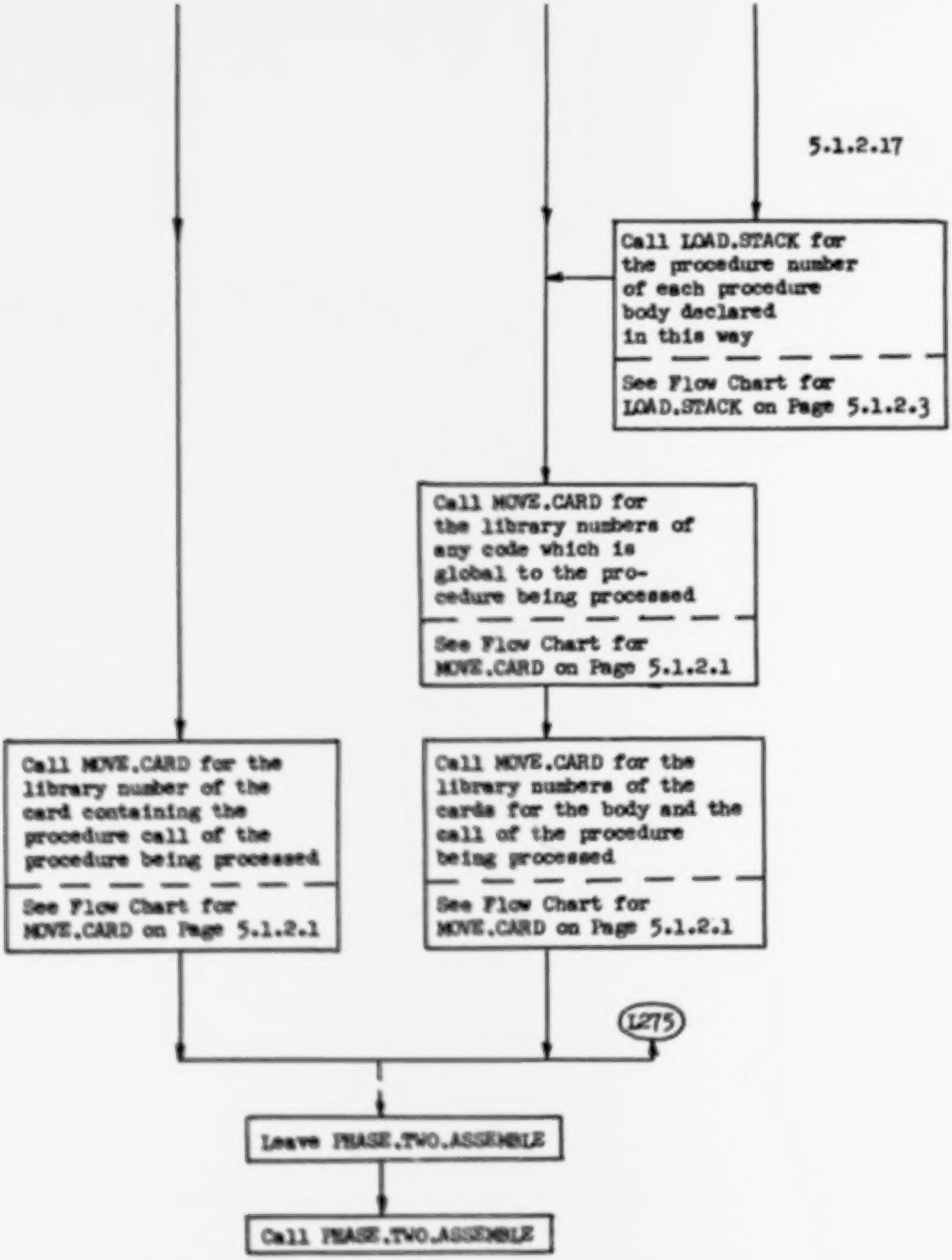
Call CHECK.STACK for each of the variables which must be defined in order for this procedure to run

See Flow Chart for CHECK.STACK on Page 5.1.2.3

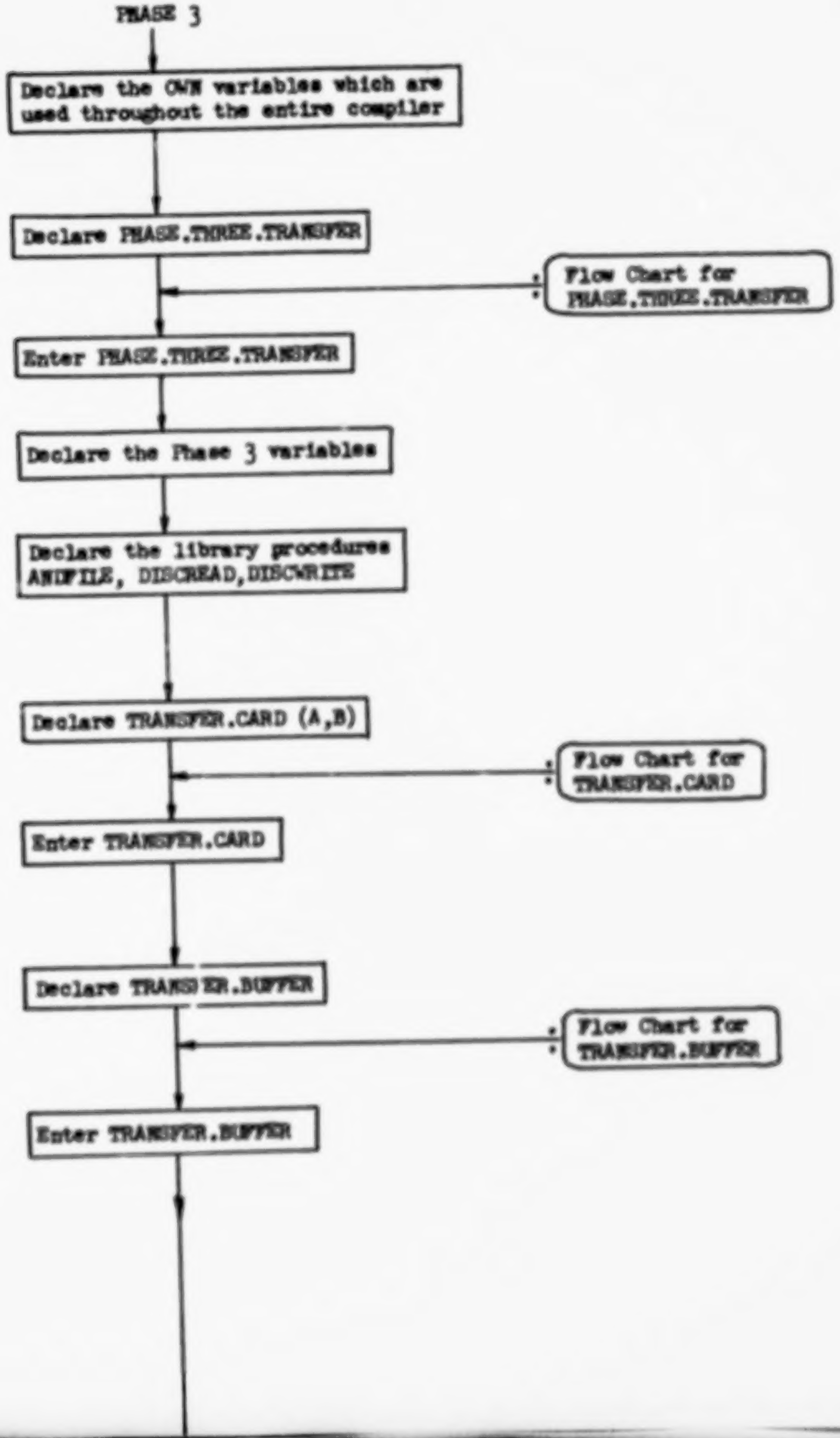
Call LOAD.STACK to place into the STACK the identifying numbers for each of the new global variables defined by this procedure

See Flow Chart for LOAD.STACK on Page 5.1.2.3

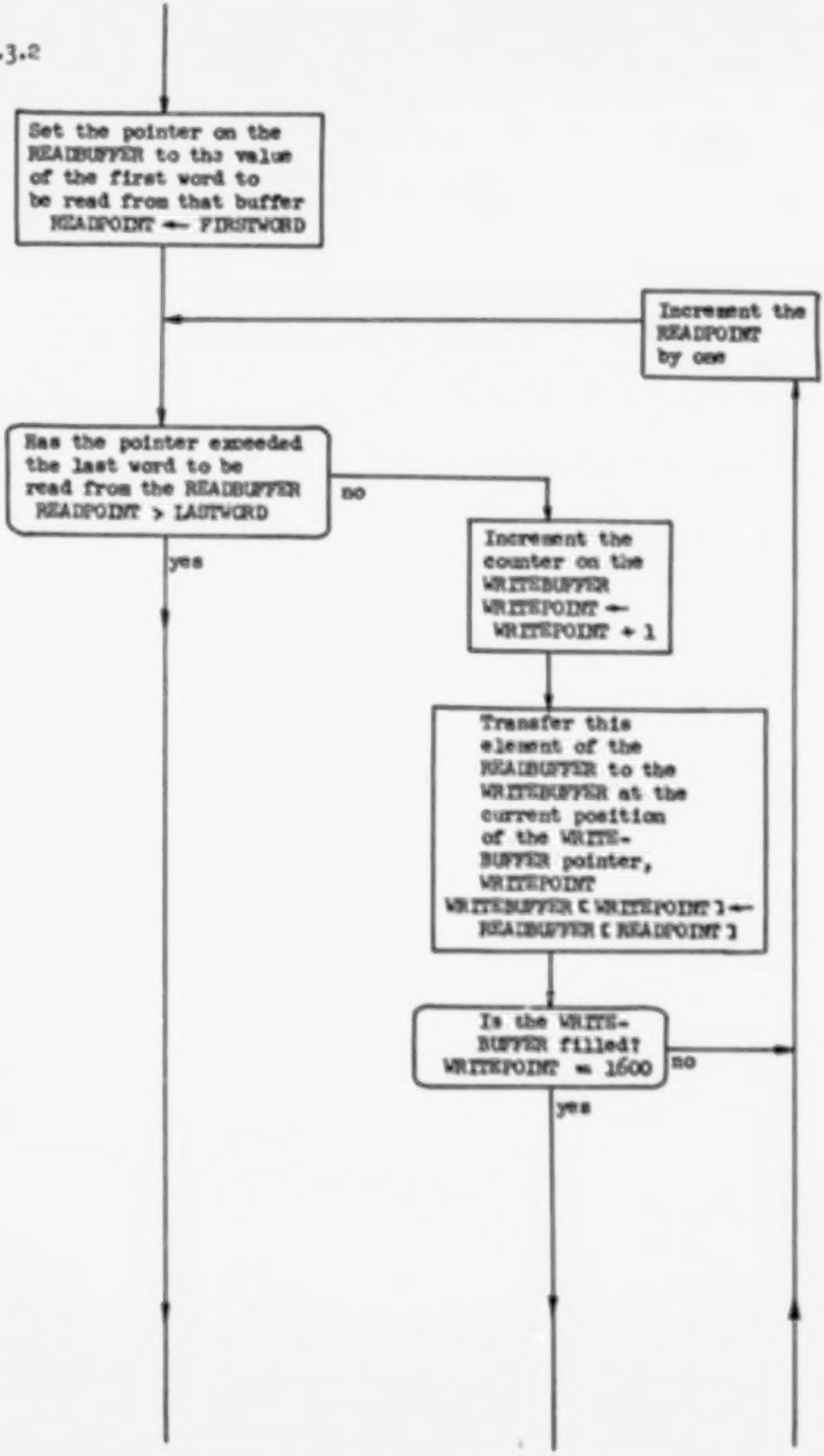

Are there other procedures used by the body of the procedure which is being processed
yes
no
Call IN.THE.STACK for each of these procedures to see if it has been declared previously
See Flow Chart for IN.THE.STACK on Page 5.1.2.2
Have they been declared
yes
no
Call MOVE.CARD for the library numbers of any code which must be global to each of these procedures
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Call MOVE.CARD for the library numbers of the code for the bodies of each of these procedures
See Flow Chart for MOVE.CARD on Page 5.1.2.1

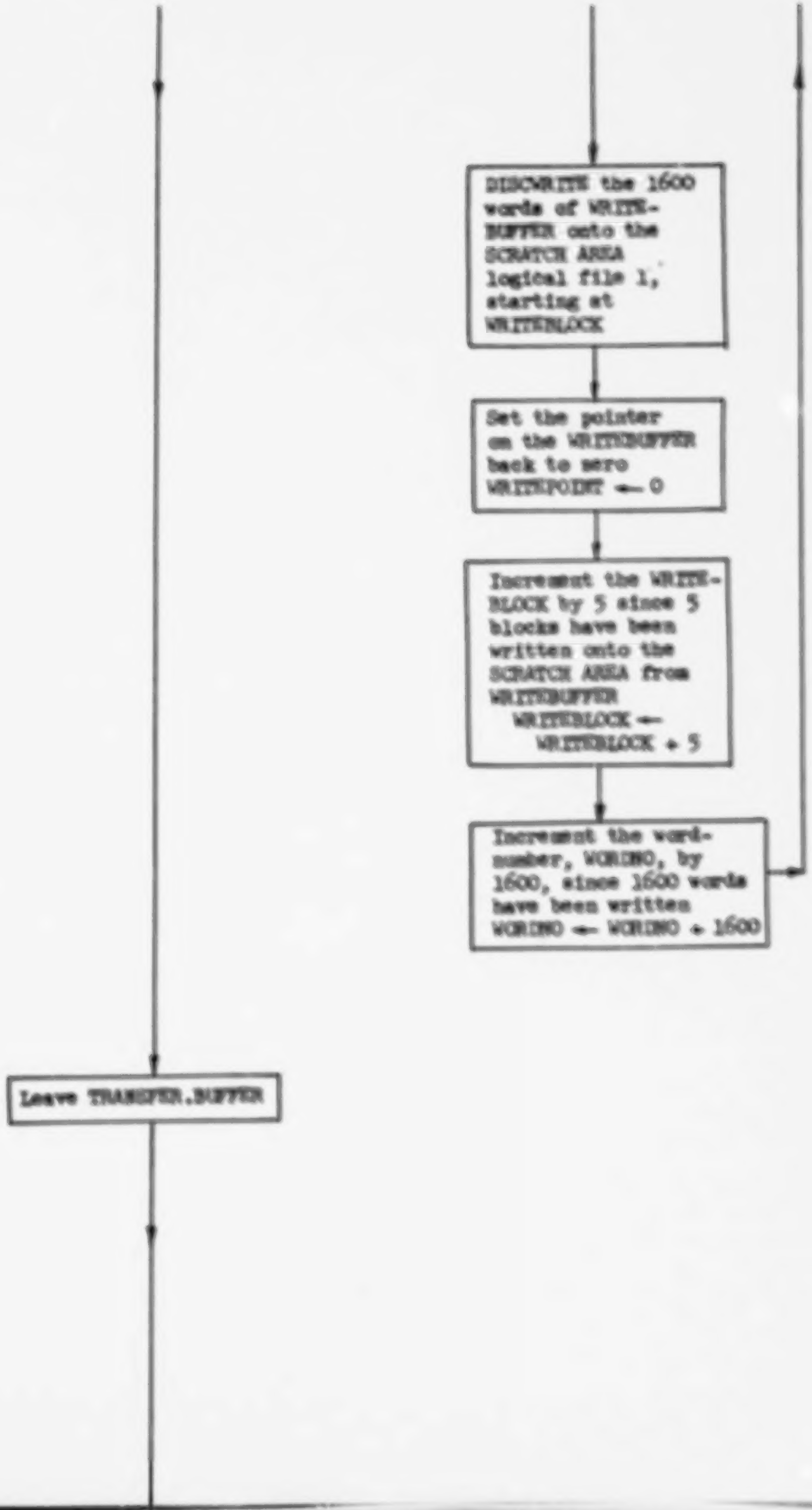

Call LOAD.STACK for the procedure number of each procedure body declared in this way
See Flow Chart for LOAD.STACK on Page 5.1.2.3
Call MOVE.CARD for the library numbers of any code which is global to the procedure being processed
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Call MOVE.CARD for the library number of the card containing the procedure call of the procedure being processed
See Flow Chart for MOVE.CARD on Page 5.1.2.1
Call MOVE.CARD for the library numbers of the cards for the body and the call of the procedure being processed
See Flow Chart for MOVE.CARD on Page 5.1.2.1
L275
Leave PHASE.TWO.ASSEMBLE
Call PHASE.TWO.ASSEMBLE

5.1.3 Phase Three - Transferral

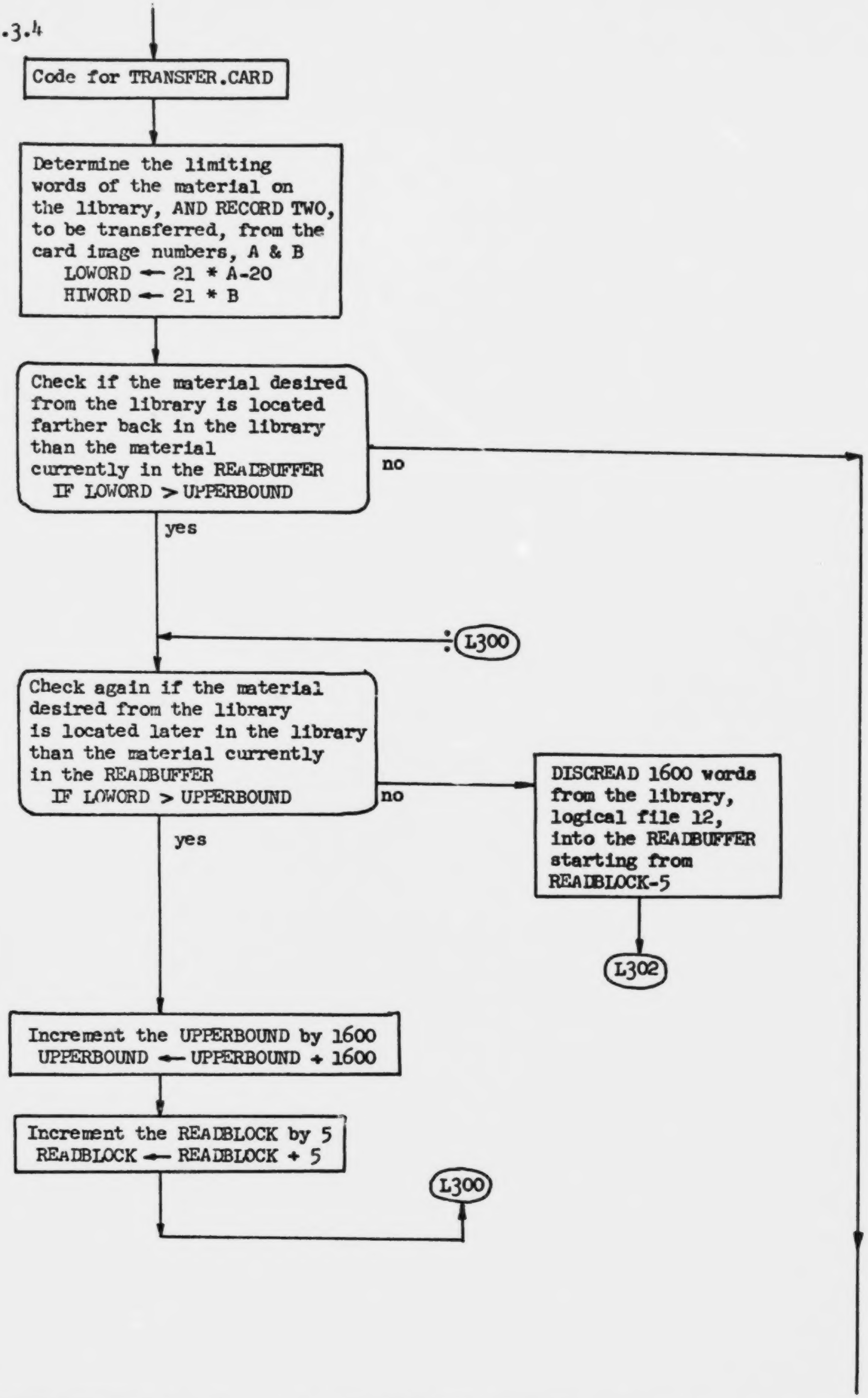

PHASE 3
Declare the OWN variables which are used throughout the entire compiler
Declare PHASE.THREE.TRANSFER
Flow Chart for PHASE.THREE.TRANSFER
Enter PHASE.THREE.TRANSFER
Declare the Phase 3 variables
Declare the library procedures ANDFILE, DISCREAD, DISCWRITE
Declare TRANSFER.CARD (A,B)
Flow Chart for TRANSFER.CARD
Enter TRANSFER.CARD
Declare TRANSFER.BUFFER
Flow Chart for TRANSFER.BUFFER
Enter TRANSFER.BUFFER

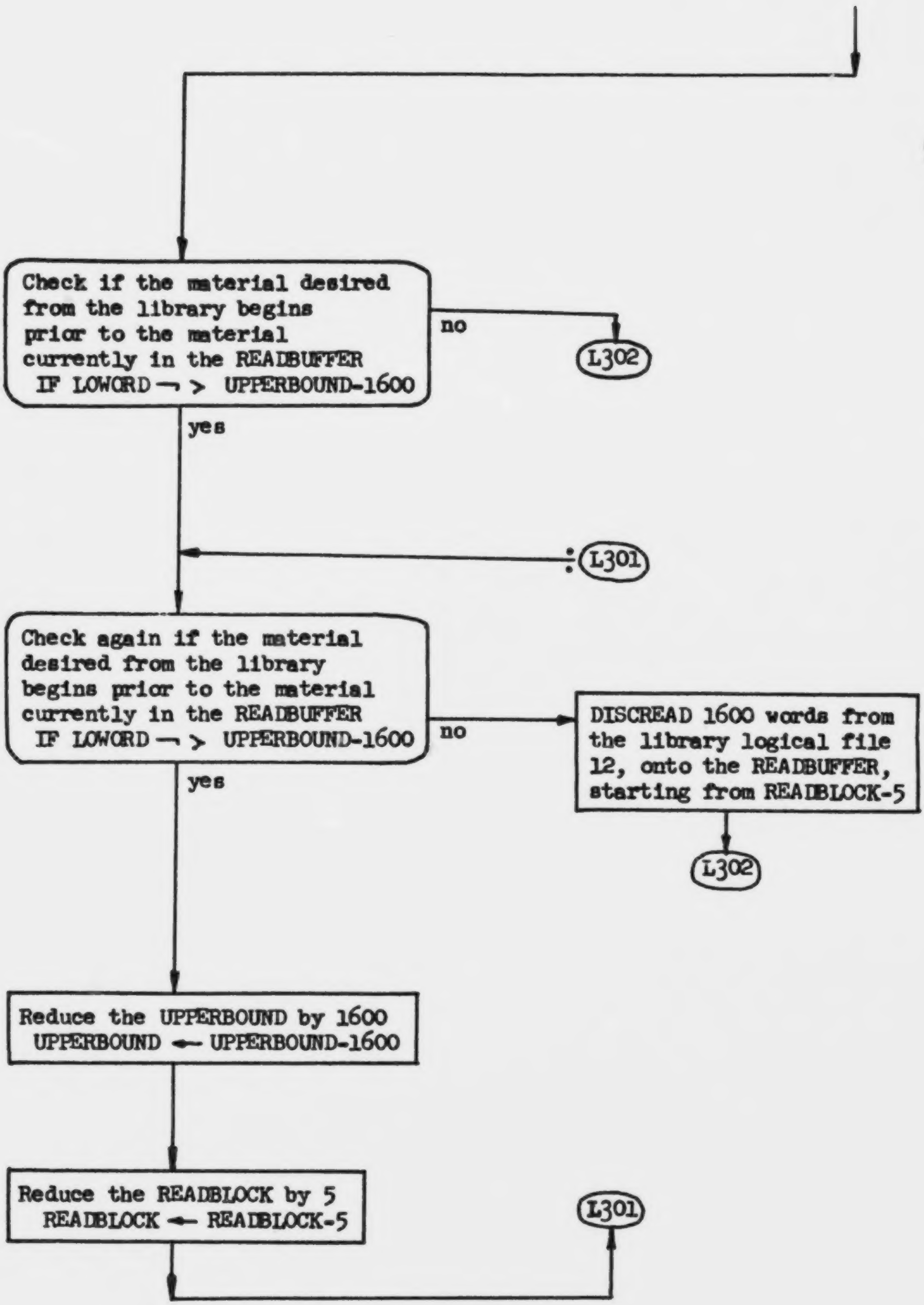
5.1.3.2

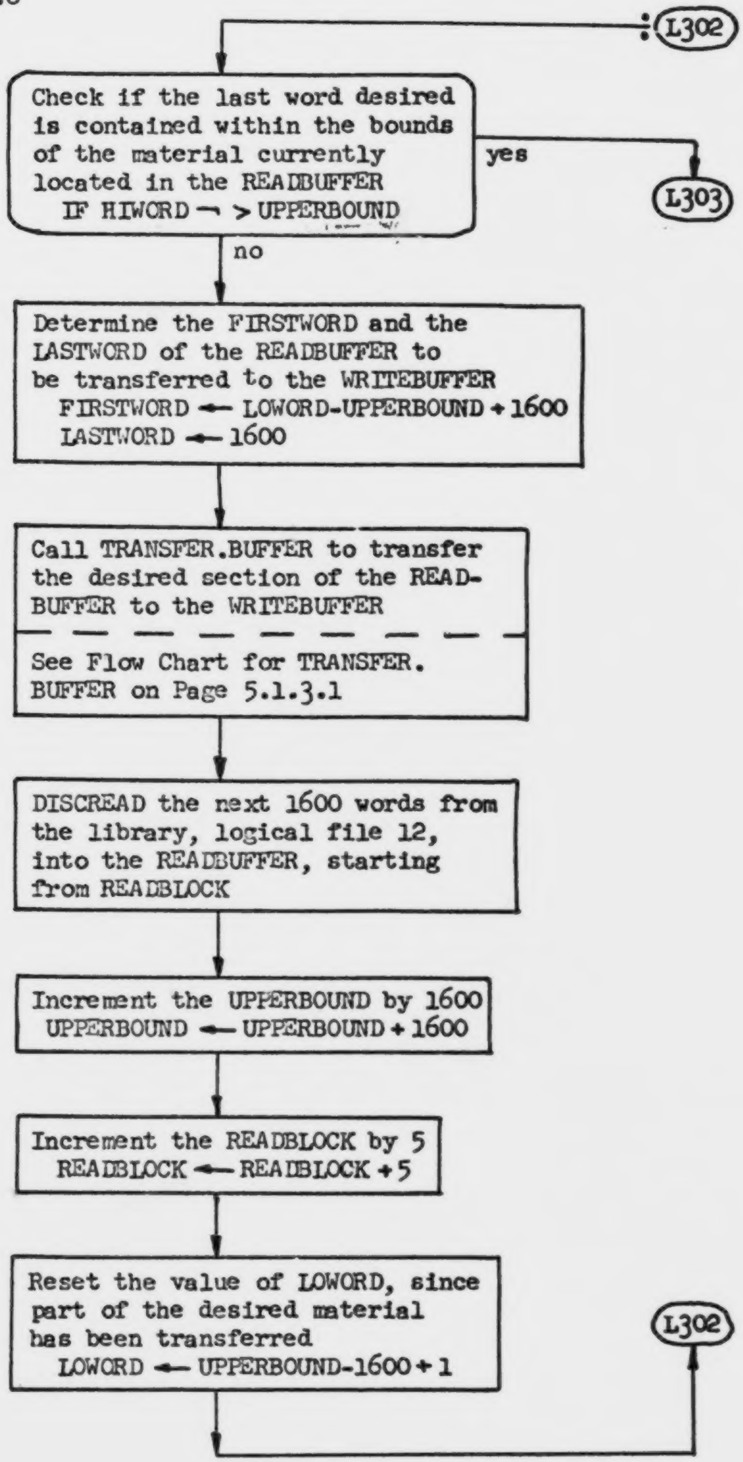

Set the pointer on the READBUFFER to the value of the first word to be read from that buffer
READPOINT ← FIRSTWORD
Increment the READPOINT by one
Has the pointer exceeded the last word to be read from the READBUFFER
READPOINT > LASTWORD
no
yes
Increment the counter on the WRITEBUFFER
WRITEPOINT ← WRITEPOINT + 1
Transfer this element of the READBUFFER to the WRITEBUFFER at the current position of the WRITE-BUFFER pointer, WRITEPOINT
WRITEBUFFER [WRITEPOINT] ← READBUFFER [READPOINT]
Is the WRITE-BUFFER filled?
WRITEPOINT = 1600
no
yes


DISCWRITE the 1600 words of WRITE-BUFFER onto the SCRATCH AREA logical file 1, starting at WRITEBLOCK
Set the pointer on the WRITEBUFFER back to zero WRITEPOINT ← 0
Increment the WRITE-BLOCK by 5 since 5 blocks have been written onto the SCRATCH AREA from WRITEBUFFER WRITEBLOCK ← WRITEBLOCK + 5
Increment the word-number, WORDNO, by 1600, since 1600 words have been written WORDNO ← WORDNO + 1600
Leave TRANSFER.BUFFER

5.1.3.4


Code for TRANSFER.CARD
Determine the limiting words of the material on the library, AND RECORD TWO, to be transferred, from the card image numbers, A & B
LOWORD ← 21 * A-20
HIWORD ← 21 * B
Check if the material desired from the library is located farther back in the library than the material currently in the READBUFFER
IF LOWORD > UPPERBOUND
no
yes
L300
Check again if the material desired from the library is located later in the library than the material currently in the READBUFFER
IF LOWORD > UPPERBOUND
no
yes
DISCREAD 1600 words from the library, logical file 12, into the READBUFFER starting from READBLOCK-5
L302
Increment the UPPERBOUND by 1600
UPPERBOUND ← UPPERBOUND + 1600
Increment the READBLOCK by 5
READBLOCK ← READBLOCK + 5
L300


Check if the material desired from the library begins prior to the material currently in the READBUFFER
IF LOWORD ¬> UPPERBOUND-1600
no
L302
yes
L301
Check again if the material desired from the library begins prior to the material currently in the READBUFFER
IF LOWORD ¬> UPPERBOUND-1600
no
DISCREAD 1600 words from the library logical file 12, onto the READBUFFER, starting from READBLOCK-5
L302
yes
Reduce the UPPERBOUND by 1600
UPPERBOUND ← UPPERBOUND-1600
Reduce the READBLOCK by 5
READBLOCK ← READBLOCK-5
L301



Check if the last word desired is contained within the bounds of the material currently located in the READBUFFER
IF HIWORD ¬ > UPPERBOUND

Determine the FIRSTWORD and the LASTWORD of the READBUFFER to be transferred to the WRITEBUFFER
FIRSTWORD ← LOWORD-UPPERBOUND + 1600
LASTWORD ← 1600

Call TRANSFER.BUFFER to transfer the desired section of the READBUFFER to the WRITEBUFFER

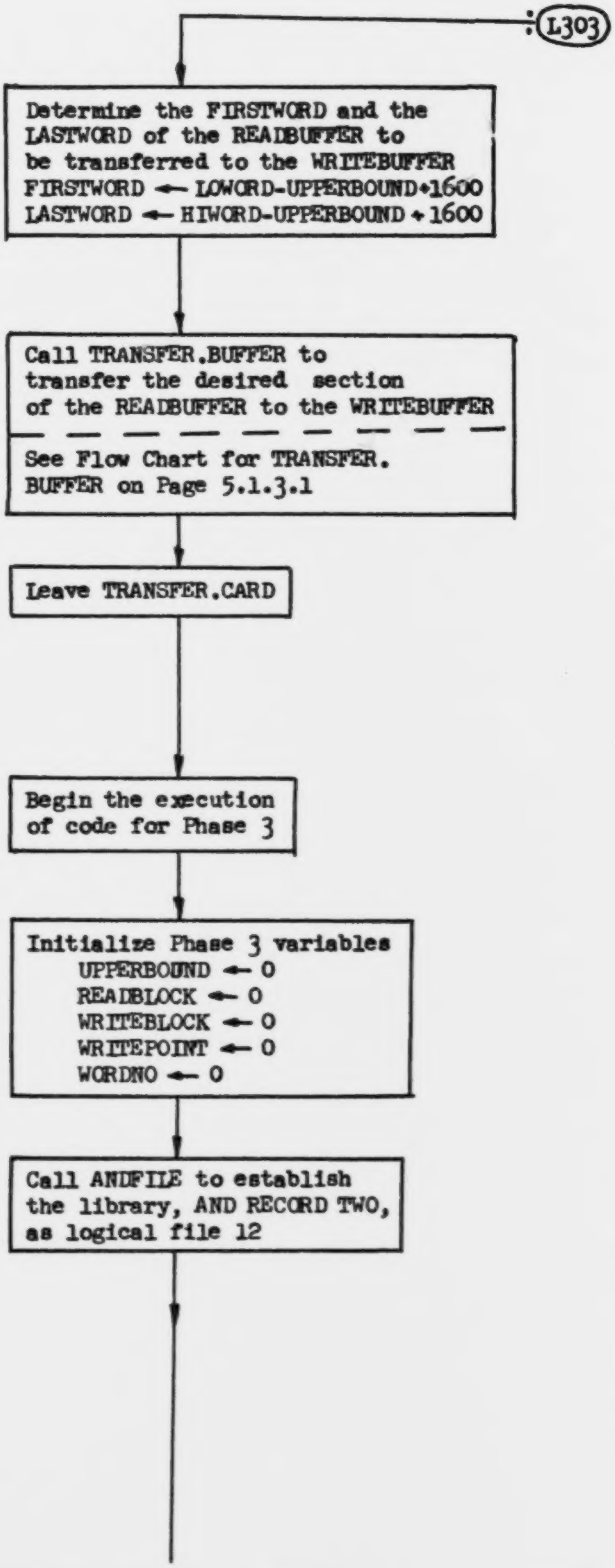
See Flow Chart for TRANSFER. BUFFER on Page 5.1.3.1

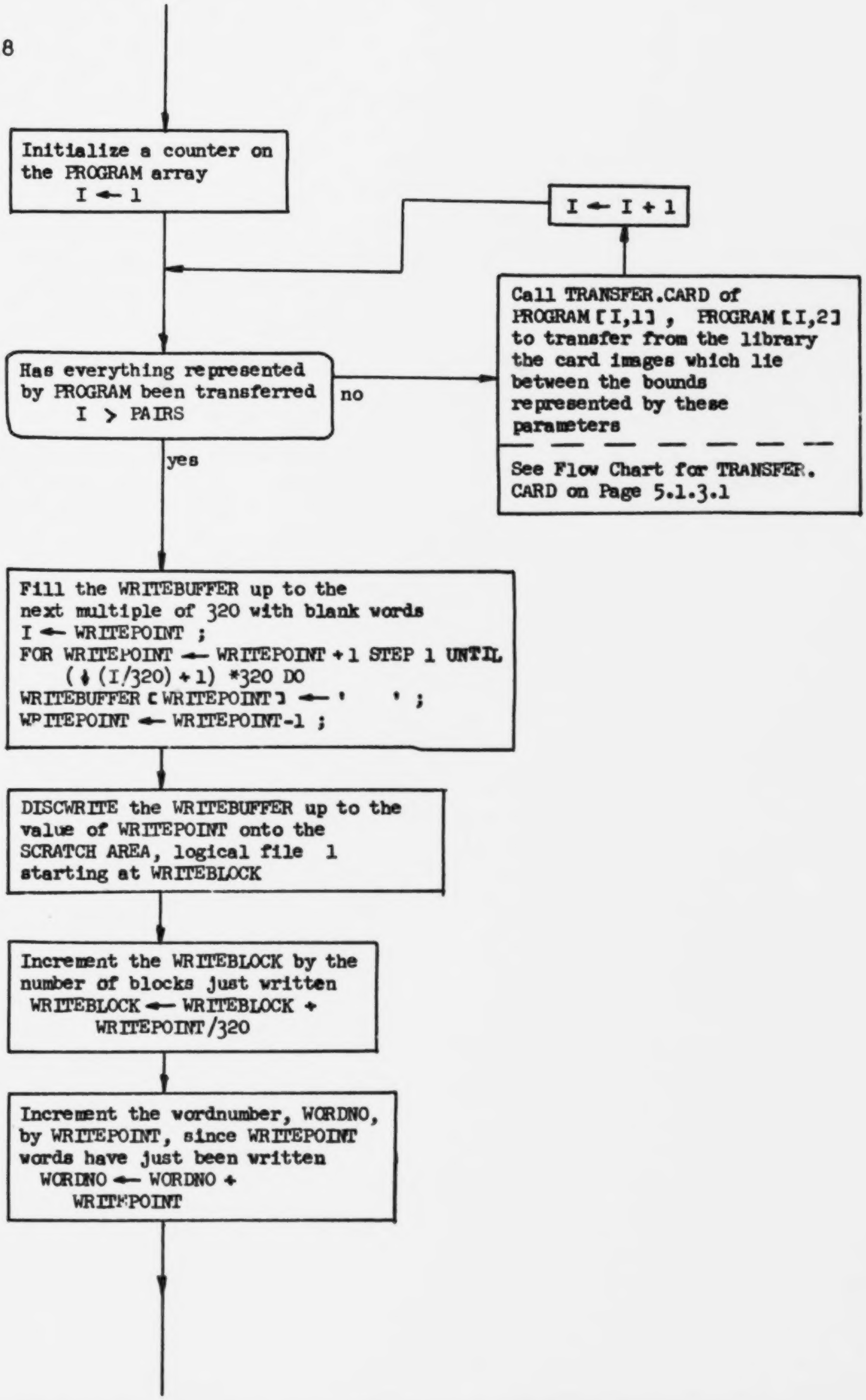
DISCREAD the next 1600 words from the library, logical file 12, into the READBUFFER, starting from READBLOCK

Increment the UPPERBOUND by 1600
UPPERBOUND ← UPPERBOUND + 1600

Increment the READBLOCK by 5
READBLOCK ← READBLOCK + 5

Reset the value of LOWORD, since part of the desired material has been transferred
LOWORD ← UPPERBOUND-1600 + 1


L303
Determine the FIRSTWORD and the LASTWORD of the READBUFFER to be transferred to the WRITEBUFFER
FIRSTWORD ← LOWORD-UPPERBOUND+1600
LASTWORD ← HIWORD-UPPERBOUND + 1600
Call TRANSFER.BUFFER to transfer the desired section of the READBUFFER to the WRITEBUFFER
See Flow Chart for TRANSFER. BUFFER on Page 5.1.3.1
Leave TRANSFER.CARD
Begin the execution of code for Phase 3
Initialize Phase 3 variables
UPPERBOUND ← 0
READBLOCK ← 0
WRITEBLOCK ← 0
WRITEPOINT ← 0
WORDNO ← 0
Call ANDFILE to establish the library, AND RECORD TWO, as logical file 12



Initialize a counter on the PROGRAM array
I ← 1

Has everything represented by PROGRAM been transferred
I > PAIRS

Call TRANSFER.CARD of PROGRAM [I,1] , PROGRAM [I,2] to transfer from the library the card images which lie between the bounds represented by these parameters

See Flow Chart for TRANSFER. CARD on Page 5.1.3.1

I ← I + 1

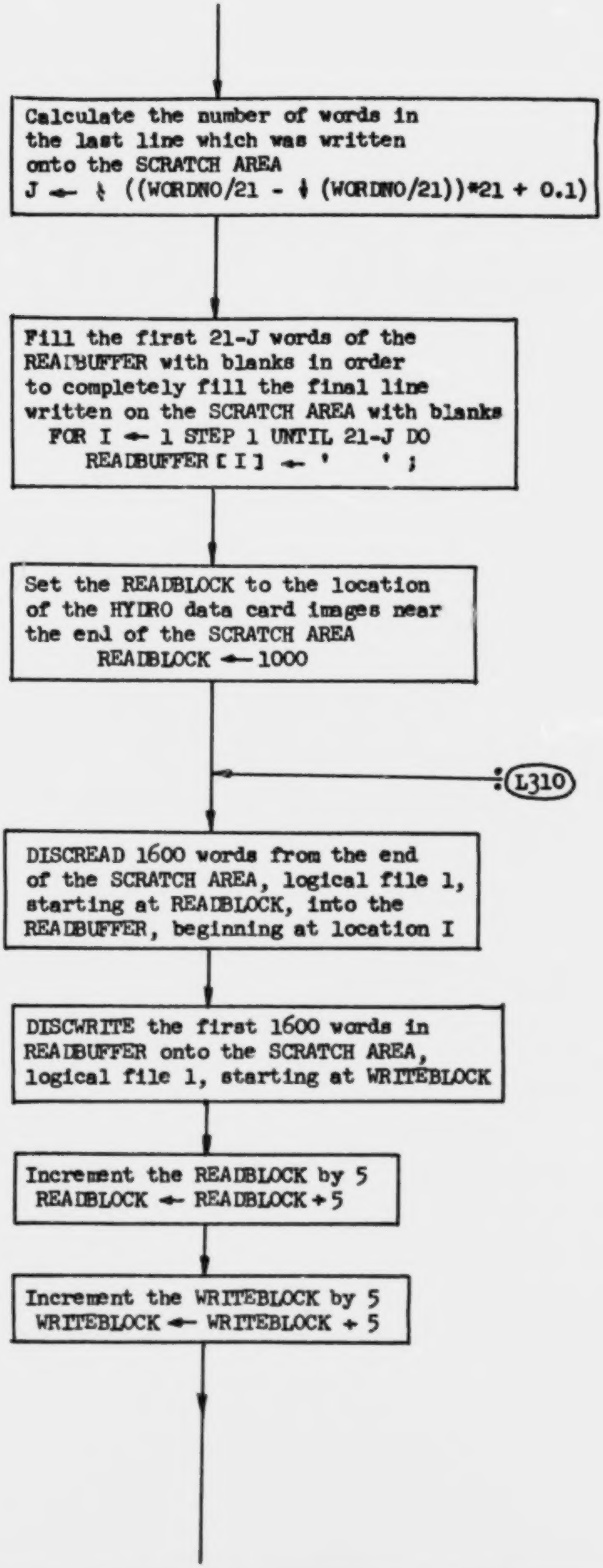
Fill the WRITEBUFFER up to the next multiple of 320 with blank words

```
I ← WRITEPOINT ;
FOR WRITEPOINT ← WRITEPOINT + 1 STEP 1 UNTIL
    ( ↓ (I/320) + 1) *320 DO
WRITEBUFFER [ WRITEPOINT ] ← '     ' ;
WRITEPOINT ← WRITEPOINT-1 ;
```

DISCWRITE the WRITEBUFFER up to the value of WRITEPOINT onto the SCRATCH AREA, logical file 1 starting at WRITEBLOCK

Increment the WRITEBLOCK by the number of blocks just written
WRITEBLOCK ← WRITEBLOCK + WRITEPOINT/320

Increment the wordnumber, WORDNO, by WRITEPOINT, since WRITEPOINT words have just been written
WORDNO ← WORDNO + WRITEPOINT



Calculate the number of words in the last line which was written onto the SCRATCH AREA

```
J ← ↓ ((WORDNO/21 - ↓ (WORDNO/21))*21 + 0.1)
```

Fill the first 21-J words of the READBUFFER with blanks in order to completely fill the final line written on the SCRATCH AREA with blanks

```
  FOR I ← 1 STEP 1 UNTIL 21-J DO
     READBUFFER [ I ] ← '   ' ;
```

Set the READBLOCK to the location of the HYDRO data card images near the end of the SCRATCH AREA

```
      READBLOCK ← 1000
```

DISCREAD 1600 words from the end of the SCRATCH AREA, logical file 1, starting at READBLOCK, into the READBUFFER, beginning at location I

DISCWRITE the first 1600 words in READBUFFER onto the SCRATCH AREA, logical file 1, starting at WRITEBLOCK

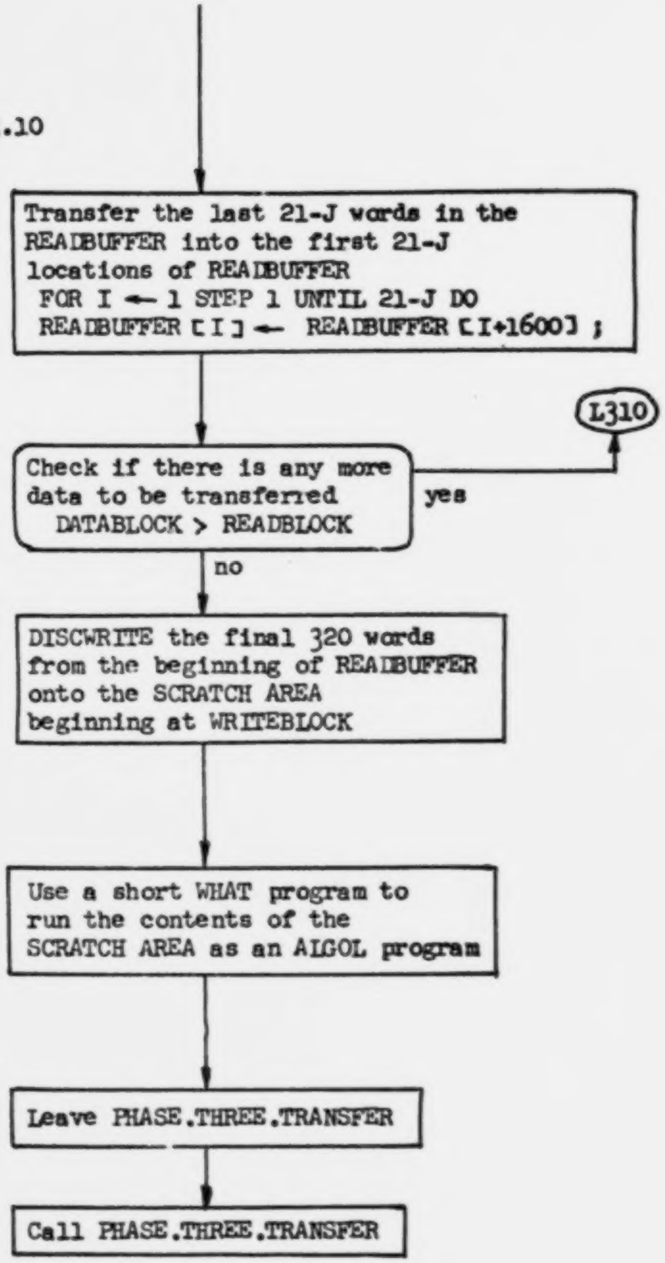
Increment the READBLOCK by 5

```
 READBLOCK ← READBLOCK + 5
```

Increment the WRITEBLOCK by 5

```
 WRITEBLOCK ← WRITEBLOCK + 5
```

5.1.3.10


Transfer the last 21-J words in the READBUFFER into the first 21-J locations of READBUFFER
FOR I ← 1 STEP 1 UNTIL 21-J DO
READBUFFER [I] ← READBUFFER [I+1600] ;
Check if there is any more data to be transferred
DATABLOCK > READBLOCK
yes
L310
no
DISCWRITE the final 320 words from the beginning of READBUFFER onto the SCRATCH AREA beginning at WRITEBLOCK
Use a short WHAT program to run the contents of the SCRATCH AREA as an ALGOL program
Leave PHASE.THREE.TRANSFER
Call PHASE.THREE.TRANSFER

5.2 Explanation of the Flow Chart

5.2.1 Phase One - Recognition

The HYDRO input program causes an automatic LINK to the first segment of Phase One, which is then run in computer core as an ALGOL program.

Variables whose values are not to be lost when LINKs are made to other segments of the compiler are declared as OWN variables at the beginning of Phase 1. Their declaration must occur before PHASE.ONE.RECOGNIZE is declared, since the OWN variables are global to it, and will remain defined throughout the execution of the compile programs.

PHASE.ONE.RECOGNIZE is then declared. The variables used within it are declared, as well as two library procedures, DISCWRITE and LINK. Then the four procedures used within PHASE.ONE are declared: NEXT.COLUMN, PR, SR, and AR.

NEXT.COLUMN begins by incrementing the counter on the card columns, I, by one. Then a check is made to see if the end of the card has been reached. If it has, the next HYDRO card image is read alphanumerically into the 80 locations of the IMAGE vector by means of NAME and READ statements. More information is available on NAME and READ statements in Section 4 of Appendix I. The eighty-first location of the IMAGE vactor has an asterisk placed into it to make certain a comma or semicolon is not located there. This eighty-first element of IMAGE is required for the proper functioning of some of the Phase 1 code even though it does not exist on the actual input card. The Boolean switch STORED is then set to FALSE so that near the end of Phase 1 this card image, if it is a data card, will be stored into the DATABUFFER prior to transferral to the end of the SCRATCH AREA. The card column pointer is then set to one, and an exit is made from NEXT.COLUMN.

If it is found at the beginning of NEXT.COLUMN that the end of a card has not been reached, an exit is immediately made from the procedure.

Following NEXT.COLUMN, the procedure PR is declared. After proper declaration statements, the variable LETTER is set to one. LETTER represents the character of the IMAG vector which is currently being compared to one of the parameters of PR. The first parameter of PR, D1, is then assigned to the variable PARAMETER, and a comparison is made. If a match is found, i.e. if PARAMETER equals IMAG [LETTER], then the LETTER veriable is incremented by one, and the next parameter is stored into PARAMETER by means of the iteration statement. If the parameter list has not been exhausted, another comparison is made is search of a match. This process continues as long as anything remains in the parameter list and matchings are found. Whenever a match is not found, control passes to label L100. If on the other hand, matchings are always found, and the parameter list is exhausted, an asterisk is stored into the PARAMETER variable in order to carry out the check which is to take place at L100. The check at L100 is made to see if a complete matching has been found between the contents of IMAG and the parameters of PR. If all the characters in IMAG have been scanned, a dollar sign will be located in IMAG [LETTER]. If all of the letters or numbers in the parameter list have been matched with IMAG, an asterisk will be located in PARAMETER. Thus, if both these conditions are met, the identifying number for the procedure represented in the PR call can be placed into the STACK. Before this is done, a check is made to determine if data were located on this card IMAGE prior to the procedure call. This error will have occurred if the STORED variable is TRUE, and an exit can be made to the error message at L197. If this error is not found, PROCBASE plus the number parameter of

PR, N, is placed into the STACK at the current STACK position, STNO. A ckeck has to be made at this point to see if the procedure call recognized was PROGRAM.END. If it was, only the value of PROCBASE will be in the STACK, since N is zero for the PROGRAM.END call on PR. When this occurs, an exit is made to L199. If this is not the case, the stack number is incremented by one, since one element has been added to it. Following this, two variables are initialized in preparation for the recognition of the next procedure or data variable. The TRANSFER variable is set to 1 to indicate that a new run through all the Phase One segments can be made to recognize the next procedure or variable, and SKIPOVER is made FALSE so that Phase One will not skip past the code for storing the next procedure or variable name into IMAG. After this the card column pointer is set to 80 so that a new card will be read into IMAGE.

Control then passes to L110 to process the next HYDRO input card. If, on the other hand, a match is not found between IMAG and PR, an exit is simply made from the PR procedure.

The declarations of SR and AR follow the declaration of PR. A glance at the compiler programs will show that these work in a similar way to PR except for the following minor details. PR has 30 character parameters, whereas SR and AR have only 15. The SR and AR procedures do not have to check for PROGRAM.END as PR did. If a match is found, the SR procedure stores N into the STACK; while AR stores ARRAYBASE + N. These two procedures set the column pointer to the column on the data card containing the left arrow rather than the last column on the card. They also set the TYPEDATA variable to blanks to indicate that the type of data on the card has not been ascertained yet. The procedures then exit to L150, where code is located which will skip over the data and reach the next variable or procedure name.

After FR, SR, and AR are declared, the actual code of Phase 1 which makes use of these procedures is executed. This code begins by checking if Phase One has just been entered and the Phase One variables need to be initialized. ARRAYBASE not being equal to 10000 indicates that these variables have not been initialized. The purpose of each of these variables is explained in Chapter Four and their initial values are self-explanatory. As part of this initialization, the card image pointer, I, is set to 80 so that the first HYDRO input card will be read into IMAGE at L110.

At label L110 a call is made on NEXT.COLUMN. This increments the column pointer, and if the end of the card has been reached, reads a new card into IMAGE. A check is then made on SKIPOVER to see if a procedure or variable has already been loaded into IMAG so that the code which follows can be skipped. If SKIPOVER is TRUE, i.e., a LINK has been made to this segment of Phase One from another segment which loaded IMAG, then control passes to label L115. If, on the other hand, IMAG must be filled, execution continues at L113. NEXT.COLUMN is called, and if the new card column is empty, a return is made to L113 to move to the following column. This process continues until something is found on a card. When the first letter of a procedure call or data variable is found, I represents the column in which it is located.

I is stored into J, and an iteration is started to move J across the card image until a blank, a left arrow, a left bracket, or the end of the card is found. If the end of the card is reached before a procedure call or a variable name terminates, an error message is given at L180. If the end of the card is not reached, then the procedure call or variable has been completed; I marks its beginning, and J marks its end. In order to compare the call or variable located with the parameters of PR, SR, or AR later on in Phase 1, the characters surrounded by I and J are stored into the logic vector IMAG. A dollar sign is placed after the last character transferred to IMAG.

K is then given the value of J. This is done in order to execute code to finally move the column pointer over to the left arrow if a data card has been found, or to move it to the end of the card if a procedure call has been found. An iteration is begun on K to move it over to find a left arrow, left bracket, or end of card. If none of these is found in a particular column, the only other legal character which may be located there is a blank. If the column is not blank, an error is given at L181. If a column is blank, K is incremented by one by means of the iteration statement, and a left arrow, left bracket, or end of card is sought in the next column. When one of these is finally found, a check is made to see if it is a left bracket. If it is, code must be executed to move the pointer, K, past the subscripts located between the brackets to get to the left arrow. This code begins at L112 by moving K to the next column. A check is then made to see if the end of the card has been reached. If it has, an error message is given at L182. If it has not, a check is made for a blank, plus, minus, digit, or comma. If one of these is found, a return is made to L112 to examine the next column. If one of these is not found, it implies that the subscript designation has ended, and a right bracket should have been found. If a right bracket was not found, an error is given at L183. If the right bracket was found, K is incremented by one to get to the next column, and a return is made to L111. At L111 the left arrow can now be located.

When the left arrow or end of card is finally found in the iteration at L111, the code to move through the bracketted subscript designations is skipped, and a check is made to see if the end of the card has been reached. If it has been, there is a procedure call on the card, and the TYPE variable is set to 0 to indicate this. TYPE is set to 1 when the end of the card is not reached, indicating a data card.

At L115 a check is then made on the TYPE variable. If it contains a 1 control passes to L130 for the recognition of the data variable. Otherwise the code for procedure recognition is begun.

The initial step in procedure recognition is to look at IMAGE [I], the first character of the procedure call. If it is not one of the letters of the alphabet, an error is given at L184. If it is, control transfers to the appropriate label for recognizing procedure calls beginning with the letter in IMAGE [I].

If there are any calls on PR at the particular label these are executed in an attempt to match the contents of IMAG with one of the legal HYDRO procedures. If a match does happen to be found control exits from the PR procedure to L110 or L199. If, on the other hand, a match is not made, or if no calls on PR are located at the label, a check is made to see if all segments of Phase One have been run in an attempt to recognize the procedure name. If the TRANSFER counter is less than the number of Phase One segments, SEGS, then control passes to L160 from which a LINK is made to another segment of Phase One. If all segments have been checked, then the contents of IMAG are not a legal procedure, and an error is given from L185.

At the label L130 the code is located for data variable recognition. This code is carried out in the same fashion as that for procedure recognition, except that errors are given from L186 and L187, and there are calls on both SR and AR at the same label. When a match is found in either the SR or AR procedure, control exits from the procedure to L150.

At labels L150, L151, and L152, code is located for storing the data card onto the SCRATCH AREA and for skipping over the data after the left arrow and getting to the next procedure or variable name. At L150 a call is made to see if the data card has been previously transferred into the DATABUFFER, from which it will be transferred to the SCRATCH AREA. If it has been, the processing of code in the IMAGE vector continues. If it has not been stored there, it is read by means of NAME and READ statements, into the next 20 words of DATABUFFER. The following word of DATABUFFER is filled with blanks. This word is necessary for a library card number when the DATABUFFER is written onto the SCRATCH AREA. The counter on DATABUFFER is then incremented by 2 to prepare it for the storage of the next data card, and the switch is set which indicates that the card has been placed into the DATABUFFER and will subsequently be read onto the SCRATCH AREA.

A check is then made to see if the DATABUFFER is full. If not, control returns to the processing of the data card in IMAGE. If, on the other hand, the counter, DANC, equals or exceeds 1600, it is first reset to one. Then the first 1600 words in DATABUFFER are written onto the SCRATCH AREA, beginning at the block given by DATABLOCK. The DATABLOCK is incremented by 5, because 5 blocks of data were written onto the SCRATCH AREA. Since the DATABUFFER could have been filled slightly beyond 1600, the EXCESS words at its end are transferred to its beginning, so that they will be transferred the next time the DATABUFFER is emptied. Next, the variable EXCESS is reduced by 4, since every time the DATABUFFER fills beyond 1600, it has four less words than the previous time. A check is then made to see if EXCESS has become negative. If it has, it is reset to the approproate positive number to which the DATABUFFER will be filled the next time it fills beyond 1600 words. At this point , control returns to the processing of the code in the IMAGE vector. Code is executed at this point to skip over the data following the left arrow in order to reach the following procedure name or data variable. A check is first made to see if the present column of IMAGE is blank. If it is, a return is made to L150 to call NEXT.COLUMN again. This cycle back to L150 continues until the pointer on IMAGE has moved from the position of the left arrow to the position of the first piece of data following the left arrow.

When finally a character is located following the left arrow on the data card, a check is made to see if it is the beginning of a legal piece of data. If it is not a plus, minus, digit, decimal point, or asterisk, an error message is given from L188. If it is one of these, a check is made to see if it is an asterisk; this tells that a piece of alphanumeric data has been located, and a transfer is made to L152 to bypass it.

If the character is not an asterisk, then a piece of numerical data has been located. Before this data is bypassed at L151, a check is made to see if alphanumeric data has been previously given to the variable being processed. If the TYPEDATA is ALPH, then an error has been made by storing both alphanumeric and numeric data into the same variable; an error message is given at L189. If the error was not made, the TYPEDATA is designated as

NUMR. A check is then made for the plus or minus sign which may preceed a piece of numerical data. If one of these is found, a move is made to the NEXT.COLUMN. Then the variable HAVEPERIOD is set FALSE to indicate that a decimal point has not yet been found in this piece of numerical data. At this point, at L151, everything has been prepared for the recognition of the piece of numerical data itself.

At L151 a check is first made for a plus or minus sign. Since these were checked for previously, if another appears at this point, an error is given from L190. If a plus or minus is not found, a check is made to see if this column of IMAGE contains a digit. If it does, a check is made to see if that digit is located in the last column on the card. If it is, an error is given from L191, since a data card can only end with a comma or semicolon. If it is not, then a move is made to the NEXT.COLUMN, and control passes back up to L151 to recognize and bypass the next character in the piece of numerical data.

When the column of IMAGE does not contain a digit, a check is made for the decimal point. If a decimal point is found, a check is made to see if it is located in the final column of the card. If so, an error is printed from L191. If not, a check is made to see if another decimal point was found previously in this same piece of numerical data. If HAVEPERIOD is TRUE then an error is given from L192. If HAVEPERIOD is not TRUE, it is set to TRUE becuase the period has been located. A move is then made to the NEXT.COLUMN, and control returns to L151 for the next character of the piece of numerical data.

When finally both a digit and a decimal point are not found in the card column, this indicates that the end of the piece of numerical data has been found, and a comma or semicolon can be expected next on the data card.

An iteration statement moves across the card columns until something besides a blank is found; that something must either be a comma or a semicolon. A check is first made for the semicolon. If it is found, control returns to L110 to look for the next data variable or procedure name, since a semicolon signals the end of data for a particular variable. If it is not found,a check is made for the comma. If this is found, and the number in the STACK indicates that an array variable is being processed, control returns to L150 to bypass the next number given to the array. If these conditions are not met, an error is given from L193.

At L152 code is located which bypasses alphanumeric data on a data card. First a check is made to see if numeric data has previously been given to the data variable being processed. If TYPEDATA is NUMR, then an error is given from L194. If this is not so, then TYPEDATA is set to ALPH. Then the column counter is moved across the piece of alphanumeric data until the right asterisk is found. At each column position a check is made to see if the end of the card has been reached within the piece of alphanumeric data. If this happens, an error is given from L195. If, on the other hand, the asterisk is found, a search is begun for a comma or a semicolon. The column counter is incremented, and moved across the card by an iteration statement until something is found besides a blank space. First, a check is made to see if this is a semicolon, signalling the end of the alphanumerical data. If the semicolon is found, control passes immediately to L110 to find the next data variable or procedure call. If the semicolon is not found, a check is made to see if the variable being processed is an array and if a comma has been found, in which case control passes to L150, where the next piece of data is bypassed. If the last check is not true, an illegal character has been found, and an error is given at L196.

The code at L160 is executed prior to linking to another segment of Phase One when a procedure or variable is being recognized by means of calls of PR, SR, and AR. The TRANSFER counter is incremented by one to indicate the transfer being made, and the SKIPOVER variable is made TRUE so that in the forthcoming segment the code will not be executed which loads the procedure or variable name into IMAG.

The labels L180 to L197 contain the Phase One error messages. At each one of these labels, an appropriate message is printed, after which control passes to L198. At L198 the ERROR variable is made TRUE so that a LINK will not be made to Phase Two once an error has been found. Then the TRANSFER and SKIPOVER variables are initialized in preparation for the recognition of the next procedure or variable. Finally the column counter is set to 80 to that a new card will be read into IMAGE , and control passes back to L110.

L199 is the location of the code to be executed when the PROGRAM.END card is found. First, a final logic word consisting of four left arrows is put into DATABUFFER. This has the purpose of insuring the proper functioning of the DEFINE.SUBSCRIPTS procedure during the running of the assembled ALGOL program, after the reading of the last piece of data. Then the final 1600 words of the DATABUFFER are written onto the SCRATCH AREA beginning at DATABLOCK by the DISCWRITE routine, and the DATABLOCK is incremented by 5.

If any errors were found during the running of Phase 1, the procedure PHASE.ONE.RECOGNIZE is terminated at this point. If, on the other hand, ERROR is FALSE, several OWN variables are initialized in preparation for

5.2.1.12

Phase 2. The Phase 2 counter on the STACK, OPNO, is set back to the beginning, at STORBASE; the counter,PAIRS, on the PROGRAM array is set to zero; and the DATAREAD variable is made FALSE so that the library card numbers for the data reading procedure will be stored first in Phase Two into the PROGRAM array. When this has been done, a LINK is made to the first segment of Phase Two, and it is loaded into core and run as an ALGOL program.
This completes the declaration of PHASE.ONE.RECOGNIZE after which it is called and executed.

5.2.2 Phase Two - Assembly

Phase 2 begins by declaring the OWN variables for the entire compiler, and then PHASE.TWO.ASSEMBLE is declared. It initially declares some of its own local variables, as well as the LINK procedure. Then declarations are given for the 10 procedures used within Phase 2.

MOVE.CARD is the first delcared. Within MOVE.CARD the counter, PAIRS, for PROGRAM is first incremented by one. Then the two library card numbers, the parameters to the MOVE.CARD call,are stored into the two new locations of the PROGRAM array. This completes the code for MOVE.CARD.

The next procedure declared is IN.THE.STACK. This procedure checks if its parameter is located anywhere in the STACK vector. First, since the procedure is Boolean, it is set equal to FALSE, and an iteration is made back through the STACK in search of the parameter,A. If it is found, IN.THE.STACK is made TRUE. Otherwise, an exit is made from the procedure at L200 with the procedure FALSE.

The CHECK.STACK procedure uses IN.THE.STACK, calling it for its parameter, A. If the parameter is not found in the STACK an error message is printed. If it is found there, an exit is simply made from the procedure.

The LOAD.STACK procedure places the value given by its parameter into the STACK. It first decrements the lower pointer, STORBASE, on the STACK, and then stores its parameter into the new STACK position. Notice that the list of numbers called the STACK is added onto from both ends. Its original element is the initial value of STORBASE, i.e. 500. The numbers representing procedures and variables recognized by Phase One are stored above the 500th STACK position. This forward portion of the STACK then represents in numbers the program submitted by the user. Numbers placed into the STACK by LOAD.STACK, on the other hand, are added to the other end of this list, i.e. below the 500th position. These numbers represent variables and procedures which will be declared, defined, and retained during the running of the ALGOL program assembled from the HYDRO code.

SINGLE.VARIABLE is one of the two procedures used by the PROCESS.VARIABLES procedure. It first computes the position in the library, i.e. AND RECORD TWO, of the data read card for the single variable being processed, and then stores that card number into the PROGRAM array by means of a call on the MOVE.CARD procedure.

ARRAY.VARIABLE is the other procedure used by PROCESS.VARIABLES. It declares first within itself a smaller procedure it employs, called TEST.BOUNDS. TEST.BOUNDS has three parameters representing bounds on any HYDRO array; its purpose is to check if these bounds are defined. The procedure checks if all three of its parameters are IN.THE.STACK, i.e. if they will be defined at the proper time during the running of the final ALGOL program. If so, an exit is made from TEST.BOUNDS to L240 in the ARRAY.VARIABLE procedure. If they are not in the STACK, the ERROR switch is set TRUE and an error message printed. The procedure then exits to L250 of ARRAY.VARIABLE.

After TEST.BOUNDS is declared,the rest of the body of the ARRAY.VARIABLE procedure is declared. This begins with the declaration of a switch, ARRY. The locations of this switch contain calls on TEST.BOUNDS for each of the input data arrays of the HYDRO language.

Before switching to one of these locations, a check is made to see if the array represented by the number being processed in the STACK has ever been declared before in the program bein assembled into the PROGRAM array. If the present STACK element has been previously IN.THE.STACK, then the array is already delcared and DECLARED is made TRUE. Otherwise, it is made FALSE. After this, the switch location containing the call on TEST.BOUNDS for the present array is calculated. Then a computation is made of the location in the program library of the declaration and array read cards for this array. Finally a check is again made to see if the array has ever been declared. If it has, the procedure skips the testing of bounds at the proper ARRY location and goes to L245. If not, the bounds are tested, and an exit is made from TEST.BOUNDS to either L240 or L250.

At L240 a call is made on MOVE.CARD for the library cards containing both the array declaration and its data read. After this call control passes to L250 and out of ARRAY.VARIABLE. At L245, MOVE.CARD is called for the library cards containing only the array read, since it was already declared. Control likewise passes out of ARRAY.VARIABLE after this call.

Following ARRAY.VARIABLE, the three PROCESS.VARIABLES procedures are declared. Two of these, PROCESS.VARIABLES.EXCEPT and PROCESS.VARIABLES.UNTIL, have only minor variations from the other, PROCESS.VARIABLES. PROCESS VARIABLES begins at L206, where a move is made to the next STACK element. A check is then made to see if that STACK element represents a singel variable, i.e. it is less than the ARRAYBASE. If it is, the SINGLE.VARIABLE procedure is called, and control passes back to L206 to examine the next STACK element. If the present element of the STACK is not less than the ARRAYBASE, a check is made to see if it is less then the PROCBASE. If it is, then it represents an array, and ARRAY.VARIABLE is called. This assembles cards for the reading of the array, after which control goes back to L206. If the number was not less than the PROCBASE, then PROCESS.VARIABLES is completed, since the next procedure number in the STACK has been found.

The difference between PROCESS.VARIABLES.EXCEPT and PROCESS.VARIABLES is the following: Each time a new STACK element is reached in PROCESS.VARIABLES.EXCEPT, a check is made to see if it equals the parameter of the PROCESS procedure call. If it does, control passes down to L208, where the remainder of the data variables in the STACK before the next procedure number are skipped and not processed. Code for the reading of these variables will be assembled during the assembly of code for the body of the procedure.

PROCESS.VARIABLES.UNTIL also compares each STACK number with its parameter. When a match is found, control passes to L210. At L210, the remainder of the variable numbers in the STACK before the next procedure number are not skipped over as they are in PROCESS.VARIABLES.EXCEPT. The pointer on the STACK is merely moved back by one, and the procedure stops. This is done because a call will be made later for the PROCESS.VARIABLES procedure which will move the pointer to the next procedure number in the STACK.

This completes the declaration of the procedures used in Phase Two. After the declaration of these 10 procedures, the actual code of Phase Two is executed. First, the switch PROC is declared. This switch is used to get to the proper section of processing code for each procedure number encountered in the STACK.

Following the declaration of PROC the four BASE variables are defined. These give the numbers of the current cards on the library,AND RECORD TWO, which separate the various parts of the HYDRO language from each other, and are used in all of the MOVE.CARD calls of the sections of Phase Two procedure processing code.

A check is then made to find if the library card numbers for the data read procedures have been placed into PROGRAM; these card numbers will always be the first ones placed there. This check is made by testing the value of DATAREAD. If the data read procedures have not been assembled, DATAREAD will be FALSE, and MOVE.CARD is called to assemble them, along with the data-read error messages and the declaration of all HYDRO global single variables. Before this call on MOVE.CARD, the number zero is loaded into the STACK in order to assure the proper functioning of all calls on TEST.BOUNDS to be made during Phase Two. This must be done since zero is used as a dummy parameter in many of these calls. After the MOVE.CARD call the Phase Two operation number on the STACK, OPNO, is decremented by one because PROCESS.VARIABLES is about to be called, which will raise it again to its initial value. This procedure is then called to process any variable numbers in the STACK which may appear prior to the first procedure number there. This would occur if data were entered in the HYDRO program before the first procedure call. If this situation does not exist, the call on PROCESS.VARIABLES does not alter anything. After this call the DATAREAD varibale is made TRUE since the data reading procedures have been assembled into the ALGOL program.

The code at L275 is then executed. This code would have been executed immediately, if the DATAREAD variable was TRUE prior to the code just explained. This case prevails when LINKs are made among the segments of Phase Two.

The code at L275 is executed in preparation for a switch to one of the locations of PROC. First, the procedure number removed from the STACK the last time a procedure was processed is replaced to its former position. Then the location of PROC, to which a switch is to be made for the present procedure, is computed. TEMP1 and TEMP2 are redefined next. TEMP1 is given the number of the present STACK location, and TEMP2 is given the procedure number located there. With these numbers in temporary storage, the STACK location is set to zero. This will ensure the proper functioning of the call IN.THE.STACK of the procedure number during the processing code for the procedure. After the STACK location has been inptied, a switch is made to the proper location of PROC for the processing of the procedure.

All the procedures are processed in the same fashion, and in the flow chart an outline is given for a general procedure rather than any specific one.

Special action is taken, however, for the number representing PROGRAM.END in the STACK. This causes a switch to the label PO. At PO a check is first made to see if any errors have been found by the CHECK.STACK or TEST.BOUNDS procedures during Phase Two. If ERROR is TRUE, the compiler HALTS without proceeding to Phase Three. If, on the other hand, there are no errors, the procedure MOVE.CARD is called for the last time in Phase Two. This puts into PROGRAM the numbers representing the first six cards on the library, AND RECORD TWO. These contain enough ALGOL END statements to match all the BEGINs contained in the code previously assembled into PROGRAM. Following this a LINK is made to the segment containing Phase Three, which replaces Phase Two in core and runs as an ALGOL program.

Below the flow chart for the PO label is the flow chart for the other labels of the switch PROC. This describes the processing code for a general HYDRO procedure. First, the label may contain a LINK command to another segment of Phase Two. If this is so, that segment is loaded into core and executed from the beginning. The Phase Two segments are programmed so that nothing will be disrupted by this process. If, on the other hand, there is processing code at the PROC label, it will in most cases begin with a call on PROCESS.VARIABLES. In some cases, the EXCEPT or UNTIL version may be substituted for the original PROCESS.VARIABLES.

After all data variables are processed, a call is made on IN.THE.STACK for the procedure. If the procedure number has been processed previously, control passes to the end of the procedure processing code where a call is made on MOVE.CARD for the library card containing only the procedure call. Control then goes back to L275. If, on the other hand, the procedure has not been processed before, processing begins. First, CHECK.STACK is called for each of the variables which must be defined in order for the procedure to run correctly. After all of these have been checked, calls are made on LOAD.STACK to place into the STACK the numbers of all global variables which will be defined by the procedure, and which may be required by later procedures in the same program. After the STACK is checked and loaded, a check is made to see if any other procedures are called by the body of the procedure being presently processed. If none are called, the code can be assembled for the body of this procedure. If procedures are called, IN.THE.STACK is called to see if these procedures have been processed previously. If they have, nothing has to be done for them, and attention is returned to the principal procedure. If they have not, MOVE.CARD is first called for any code which must be placed global to them; the same procedure is then called for the bodies of these procedures. LOAD,STACK is then called to put the numbers of any procedures which have been processed into the STACK. After all the code for procedures used by the principal procedure has been assembled, it is processed itself. MOVE.CARD is first called for any code global to it, and is then called again for the body of the principal procedure, including the procedure call. This completes the processing of the procedure, and control returns to L275 to process the next procedure number in the STACK.

The declaration of all of the locations of the switch PROC completes the declaration of PHASE.TWO.RECOGNIZE. After its declaration, it is called and executed, during which a LINK is normally made to Phase Three.

5.2.3 Phase Three - Transferral

Phase Three begins with the declaration of the OWN variables for the entire compiler, and then PHASE.THREE.TRANSFER is declared. At the entrance to PHASE.THREE.TRANSFER, the Phase 3 variables are declared, as well as the library procedures ANDFILE, DISCREAD, and DISCWRITE.

Following these statements is the declaration of the principal procedure of Phase 3, TRANSFER.CARD. As it is entered, it immediately declares a smaller procedure, TRANSFER.BUFFER.

The purpose of TRANSFER.BUFFER is to transfer a desired portion of the code in the READBUFFER to the WRITEBUFFER. TRANSFER.BUFFER begins by setting READPOINT, the pointer on the READBUFFER to the value of the FIRSTWORD to be read from READBUFFER. FIRSTWORD indicates the position within the 1600 words of READBUFFER where transferral is to begin from READBUFFER to WRITEBUFFER, and LASTWORD tells the position where transferral is to cease. A check is made to see if the READPOINT has as yet exceeded the LASTWORD to be transferred. If it has, the compiler leaves TRANSFER.BUFFER. If it has not, the WRITEPOINT is incremented by one, and one element of the READBUFFER at READPOINT is transferred to the WRITEBUFFER at WRITEPOINT. A check is then made to see if the WRITEBUFFER is filled to 1600 words. If it is not, the READPOINT is incremented by one and another word is transferred. If the pointer on WRITEBUFFER is equal to 1600, a call is made on the library procedure DISCWRITE to write the 1600 words of WRITEBUFFER onto the SCRATCH AREA, logical file 1, beginning at the WRITEBLOCK. The pointer, WRITEPOINT, on the WRITEBUFFER is reset to zero, and the WRITEBLOCK is incremented by 5. Since 1600 additional words have been written onto the SCRATCH AREA, the WORDNO is incremented by 1600. With the WRITEBUFFER emptied, the READPOINT is increased by one, and a check is once again made to see if the LASTWORD of the READBUFFER has been transferred. When it is finally transferred, an exit is made from TRANSFER.BUFFER.

Following the declaration of the TRANSFER.BUFFER procedure, the code for TRANSFER.CARD begins. First, the limiting words, LOWORD and HYWORD of the code to be transferred from the library, AND RECORD TWO, are calculated. The variable LOWORD represents the position on AND RECORD TWO of the first set of four characters which is to be trasnferred. Since the parameter A is the card number on RECORD TWO of the first card to be transferred, it is multiplied by 21, since there are 21 words per card counting an extra word for the library card number. Twenty is subracted from this result to put LOWORD back to the first word on this card. HIWORD is the last four-character work to be transferred, and is 21 times the second parameter, B, which is the card number of the last card to be transferred.

Next a section of code is given which raises the UPPERBOUND and the READBLOCK to correct values, when the material in the READBUFFER, previously read from AND RECORD TWO, is below what is currently desired from AND RECORD TWO. First, a check is made to see if this condition is true. If it is not, control passes further down in TRANSFER.CARD to the line before label L301. If it is true, at L300, this check is again made. Each time this condition is found to be true, the UPPERBOUND is raised by 1600 and the READBLOCK by 5, and control returns to L300 for the check again. When finally the UPPERBOUND is not below the LOWORD, that is, the UPPERBOUND and READBLOCK correspond to the material which is desired from the library, a call is made on DISCREAD to load the 1600 words below the UPPERBOUND into the READBUFFER from logical file 12, AND RECORD TWO, beginning 5 blocks back from the current value of the READBLOCK. Control then passes to L302 to transfer a portion of this code.

At the line previous to L301, a section of code is located which lowers the UPPERBOUND and READBLOCK when it is found that the code in the READBUFFER, previously read from AND RECORD TWO,is beyond that which is currently desired from AND RECORD TWO. First, a check is made to see if this is the case. If it is, at L301, the UPPERBOUND and READBLOCK are lowered by 1600 and 5 respectively. Control then goes back to L301 and the check is again made to see if they have to be lowered further. They are decremented until they finally agree with the beginning of the material desired from AND RECORD TWO, indicated by LOWORD. This 1600 words of material is then read from the AND RECORD, logical file 12, into the READBUFFER by a call on DISCREAD. Control then passes to L302 where the transferral of material takes place from the READBUFFER to the WRITEBUFFER.

When the label L302 is reached, code is located in the READBUFFER some or all of which is to be transferred to the WRITEBUFFER. First, at L302, a check is made to see if the final word of material desired from AND RECORD TWO is already located in the READBUFFER. If it is, that is, if the HIWORD is not greater than the UPPERBOUND, control passes to L303 for the final transfer from the READBUFFER in this particular call on TRANSFER.CARD. If, on the other hand, the final word desired is not currently in the READBUFFER, then material will have to be transferred until the end of the READBUFFER is reached. The next 5 blocks of RECORD TWO will have to be read into the READBUFFER by DISCREAD, and transferral will have to begin again at the beginning of the READBUFFER. This process continues until everything desired from RECORD TWO has been read into the READBUFFER, transferred to the WRITEBUFFER, and from there written onto the SCRATCH AREA.

If, at label L302, a switch is not made to L303, that is, more than one transferral from the READBUFFER will have to take place, then the FIRSTWORD and the LASTWORD to be transferred are calculated. Of course, the LASTWORD is 1600 because transferral at this point will go to the end of the READBUFFER. Once these two variables are defined, TRANSFER.BUFFER is called to carry out the transfer. Then, because more material must be read, DISCREAD is called to get the next 5 blocks, or 1600 words, of material from AND RECORD TWO to the READBUFFER. The UPPERBOUND and the READBLOCK are then incremented because of the DISCREAD, and the value of the LOWORD is changed. It is redefined as the first word of the READBUFFER, since all material up to that point has already been transferred. Control then returns to L302 for the next transferral from READBUFFER. This process continues until it is discovered that the HIWORD is bounded by the present contents of the READBUFFER. At that point, control passes to L303 for the final transferral of the particular call on TRANSFER.CARD.

At L303 the FIRSTWORD and LASTWORD to be transferred from the READBUFFER are calculated, and the transfer takes place by means of a call on TRANSFER. BUFFER. An exit is then made from the TRANSFER.CARD procedure.

Following the declaration of TRANSFER.CARD, the actual code of Phase Three is executed. First, some of the Phase Three variables are initialized to zero. Then a call is made on the ANDFILE procedure to give AND RECORD TWO the file number 12.

In the next two commands of the program, the major portion of the Phase 3 code transferral takes place. By means of an iteration statement on I from 1 to the value of PAIRS, the procedure TRANSFER.CARD is called once for each pair of AND numbers in the PROGRAM array.

When finally everything represented by the PROGRAM array has been transferred to the SCRATCH AREA, a four line segment of code is executed which fills the WRITEBUFFER with blank words until it contains some multiple of 320 words. This is done for the following reason: When the calls on TRANSFER.CARD have been completed, the WRITEBUFFER will not in general contain an even multiple of 320 new words. But, the DISCWRITE routine always writes an even multiple of 320 words. Thus blanks are placed into WRITEBUFFER up to some multiple of 320 so that what was contained in the BUFFER before will not be written again when DISCWRITE is performed.

After the iteration statement has placed blanks into the WRITEBUFFER, the value of WRITEPOINT is a multiple of 320 plus 1. This one is subtracted from WRITEPOINT, and a call is then made on DISCWRITE to write the first WRITEPOINT words of the WRITEBUFFER onto the SCRATCH AREA beginning at the WRITEBLOCK. Following the DISCWRITE call, the WRITEBLOCK is incremented by the number of blocks written, and WORDNO is increased by WRITEPOINT, the number of words which were written.

The next section of Phase Three code moves the HYDRO data card images from the end of the SCRATCH AREA to the end of the ALGOL program now assembled on the forward part of the SCRATCH AREA. First, a calculation is made using WORDNO to establish where on the present card image of the SCRATCH AREA the first data word would lie, if it were transferred at this time. The intent here is to make certain that the data card images are transferred forward on the SCRATCH AREA exactly as they appear at the end of the SCRATCH AREA, and not shifted a portion of a card one way or the other. The variable, J, which is calculated here, tells how many words are in the last line which was written onto the SCRATCH AREA. If J is 8, for example, then 21-J, or 13, blanks must be placed at the end of the assembled code on the SCRATCH AREA to complete the line before data can be moved forward from the end of the SCRATCH AREA. After J is calculated, the 21-J blanks are placed into the first 21-J words of the READBUFFER by means of an iteration statement. Then the READBLOCK is set to 1000, which is the location of the data on the SCRATCH AREA. Then at the label L310, 1600 words from this area are read into the READBUFFER. The first 1600 words of READBUFFER are immdiately written at the end of the ALGOL program on the SCRATCH AREA, and the READBLOCK and WRITEBLOCK are incremented by 5. The extra words at the end of the READBUFFER are then transferred to its first 21-J locations. A check is then made to see if the data stored at the end of the SCRATCH AREA exceeds the amount of material read from there so far, that is, if DATABLOCK is grater the READBLOCK. If so, more data must be read, and control returns to L310. If enough data has been read, then a final 320 words of DATABUFFER are written onto the SCRATCH AREA by DISCWRITE. At this point, the program assembled onthe SCRATCH AREA is complete, and represents the HYDRO program submitted by the user.

A short program in the Carnegie Tech WHAT language runs the ALGOL program from the SCRATCH AREA. More information on the WHAT language and the bar routines is available in Section 9 of Appendix A. The bar-15 routine tells the computer that the SCRATCH AREA is to be the next source of code. The NAM command says it is to be run as ALGOL, and finally the bar-23 routine puts the ALGOL program into computer core and runs it.

This completes the declaration of the PHASE.THREE.TRANSFER procedure; after its declaration it is executed.

6. DATA READING PROCEDURES AND VARIABLES

This Chapter identifies the seven data reading procedures and the variables used by them. In each of the two sections of the Chapter a list is first given of procedures or variables presented there, followed by the actual explanations or definitions. These can be used in conjunction with the flow chart in the following Chapter to gain a complete understanding of the data read procedures.

6.1 Procedures

NEXT.COLUMN
NUMERICAL.DATA
ALPHA.NUMERICAL.DATA
DEFINE.SUBSCRIPTS
DATA.INPUT
SINGLE.READ
ARRAY.READ

NEXT.COLUMN - NEXT.COLUMN is a short procedure used throughout the other data read procedures. It is the data card scanner, and as such is almost identical to the NEXT.COLUMN procedure of Phase One. Each time it is called, it moves the pointer, COLM, on the logic vector, CHAR, one column to the right so that the contents of that column can be examined. When the end of a card image is reached, it reads alphanumerically the following HYDRO data card into CHAR and resets the pointer to one.

NUMERICAL.DATA - NUMERICAL.DATA is a procedure employed by both DATA.INPUT and DEFINE.SUBSCRIPTS whenever numerical data is encountered on a data card. It is a REAL procedure which examines the digits of any number one by one, calculates the correct value of the number, and then stores that value into itself.

ALPHA.NUMERICAL.DATA - The ALPHA.NUMERICAL.DATA procedure is also used by DATA.INPUT. It is called when an asterisk signifies that alphanumerical data has been found on a data card. ALPHA.NUMERICAL.DATA reads that data into the logic vector ALPH from which it is stored by SINGLE.READ or ARRAY.READ into the proper logic variable of the ALGOL program which is being executed.

DEFINE.SUBSCRIPTS - The DEFINE.SUBSCRIPTS procedure initializes the three subscript variables, SUB1, SUB2, and SUB3, used during data reads. This initialization takes place each time a new variable is about to be read. If a subscripted data variable is the next to be read, this procedure sets SUB1, SUB2, and SUB3 to the values given by the subscripts. If the variable to be read does not have subscripts, the procedure sets the above three variables to one.

DATA.INPUT - DATA.INPUT is the principal data read procedure. It employs four of the other data procedures, NUMERICAL.DATA, ALPHA.NUMERICAL.DATA, DEFINE.SUBSCRIPTS, and NEXT.COLUMN. It is a REAL procedure which takes on the value of the data itself if numerical data is read, and is subsequently stored by SINGLE.READ or ARRAY.READ into the variable being defined.

SINGLE.READ - SINGLE.READ is the procedure which actually stores data into the single variables given as data in the input HYDRO program. Calls on SINGLE.READ are assembled into the ALGOL program at proper places by the compiler so that single variables will be read prior to the procedures for which they were given as data.

SINGLE.READ differentiates between the types of single variables, and stores the result of a call on DATA.INPUT into the proper variable name.

ARRAY.READ - ARRAY.READ stores data into the arrays given as data in the HYDRO program. Calls on ARRAY.READ are placed throughout the ALGOL program similar to those for SINGLE.READ so that the proper arrays are read prior to the procedures for which they were given as data. ARRAY.READ is more complicated than SINGLE.READ; it has to read one, two, or three dimensional arrays, and it usually reads more than one piece of data. It calls DATA.INPUT to obtain numerical data which it stores into REAL and INTEGER arrays. Calls on the same procedure load alphanumerical data into ALPH from which it is stored into ALPHANUMERICAL arrays. ARRAY.READ also handles the incrementing of subscripts while data is being read into the subsequent positions of an array.

6.2 Variables

II, JJ
CHAR[1:80]
ALPH[1:20]
COLM
SUB1
SUB2
SUB3
ABCD
EXPO
TEMP
NEGATIVE
ASTERISK
DEFINED
FINISHED

II, JJ - These are integer counters used in the ALPHA.NUMERICAL.DATA and NEXT.COLUMN procedures.

CHAR[1:80] - CHAR is a logic vector into which data card images are stored by the NEXT.COLUMN procedure. The CHAR vector is processed by several of the data read procedures in order to put the input data into the ALPH vector or the REAL procedure DATA.INPUT.

ALPH[1:20] - ALPH is a logic vector used in the ALPHA.NUMERICAL.DATA procedure. Alphanumeric data being read from a data card image are stored into ALPH prior to being placed into the logic vector being defined by the data read.

COLM - The pointer on the CHAR vector is called COLM. It tells at any time what column of the data card is being scanned.

SUB1, SUB2, and SUB3 - These three variables represent the subscripts of a general three dimensional input array. If a data read card contains a subscripted variable, i.e. numbers in brackets after the variable name tell where in the array to begin storage of data, then SUB1, SUB2, and SUB3 are given the values of the array subscripts denoted by the numbers in the brackets. If, on the other hand, a variable is not subscripted, then these three variables are set to one, and storage begins at the first element of the array.

ABCD - ABCD is a logic variable which is given, in the DEFINE.SUBSCRIPTS procedure, the first four letters of the data variable to be defined by DATA.INPUT. It is used in the ARRAY.READ procedure to insure that data are being read from the correct variable on the data card.

EXPO - EXPO is the power-of-ten counter used in the NUMERICAL.DATA procedure. It represents the power of ten which must multiply the final number in NUMERICAL.DATA in order to have the decimal point in the correct position.

TEMP - In the NUMERICAL.DATA procedure, when an integer is recognized in a particular column of CHAR, that integer value is stored into TEMP prior to being added to the current value of NUMERICAL.DATA.

NEGATIVE - NEGATIVE is a Boolean switch used in the NUMERICAL.DATA procedure. It is automatically set to FALSE at the beginning of the procedure. If a minus sign precedes the number being read from the data card, NEGATIVE is changed to TRUE. Then at the end of the procedure, if NEGATIVE is TRUE, the value in NUMERICAL.DATA is made negative.

ASTERISK - The Boolean variable ASTERISK is used in the ALPHA.NUMERICAL.DATA procedure to signal when the asterisk at the right end of a piece of alphanumerical data has been found. At the outset of the procedure ASTERISK is made FALSE. It remains FALSE as subsequent groups of four characters are scanned and stored into the locations of ALPH. The scanning of the right ASTERISK causes this variable to be set to TRUE and leads to an exit from the procedure.

DEFINED - The Boolean switch DEFINED is used in the DEFINE.SUBSCRIPTS procedure to tell whether the subscript variables have been defined. It is made FALSE at the outset of the procedure, and when some or all of the three subscript variables are defined by calls on NUMERICAL.DATA, it is changed to TRUE. If it is not made TRUE, then SUB1, SUB2, and SUB3 are all set to one.

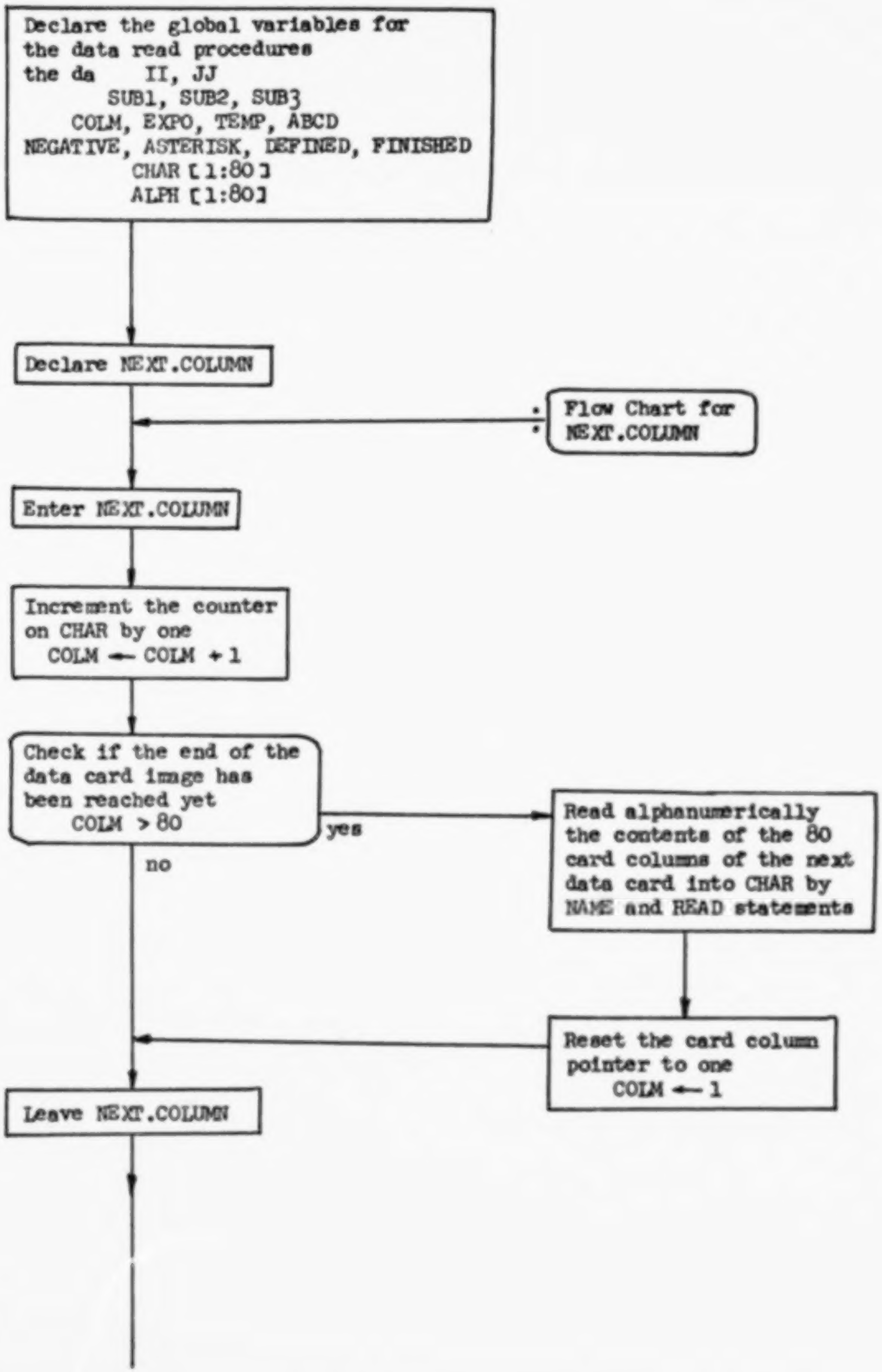
FINISHED - FINISHED is a Boolean switch which tells whether or not the reading of data for an array has been completed. It is made FALSE during the reading of data for an array or single variable, but when the semicolon is found at the end of a data read, it is set to TURE. This signal is used to terminate the execution of code for the reading of data into a particular array in the ARRAY.READ procedure.

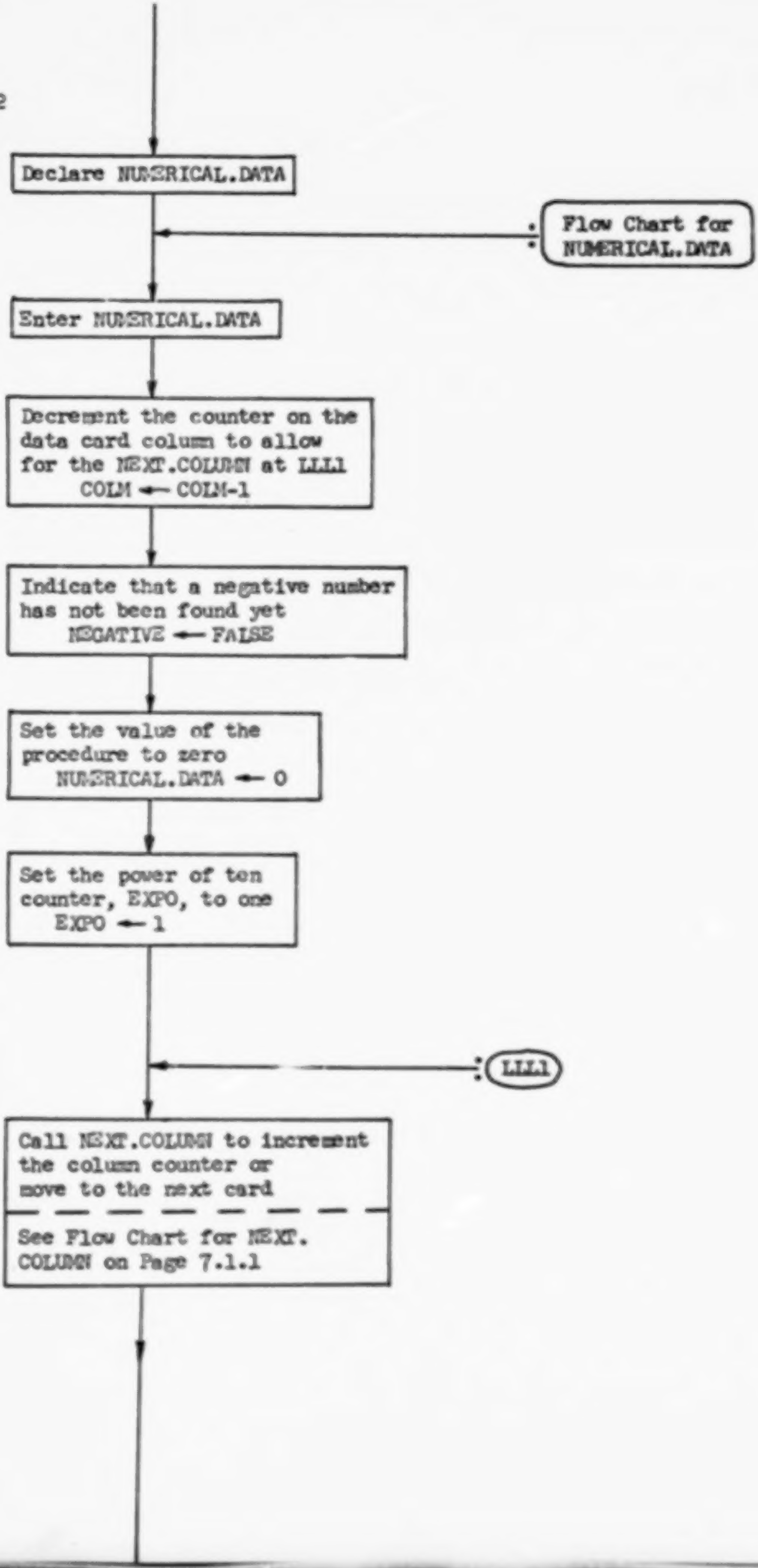
7. DETAILED DATA READ FLOW CHART

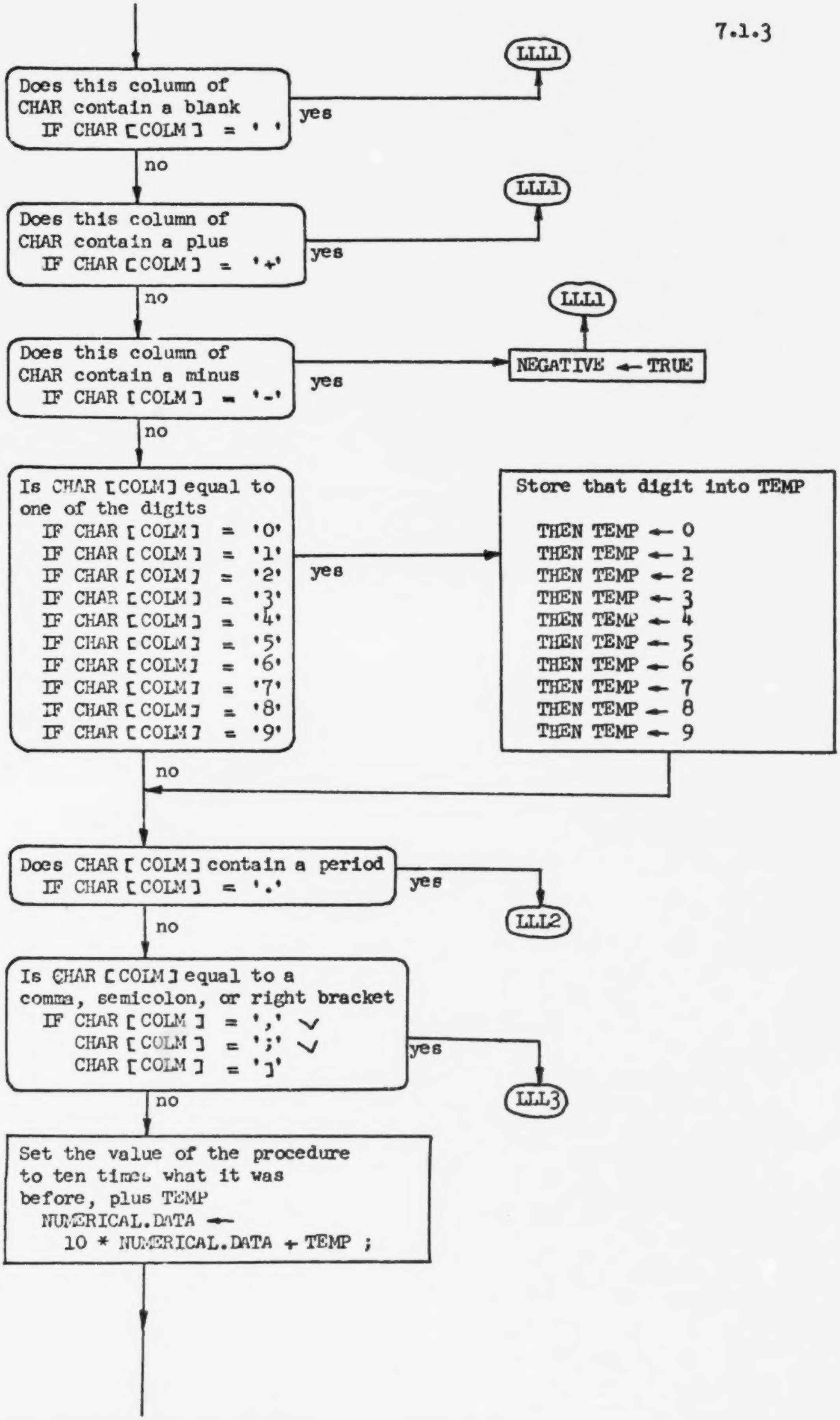
This Chapter contains the detailed flow charts and explanations of a set of procedures which are assembled at the top of every ALGOL program assembled from HYDRO. These are the seven data reading procedures, (NEXT.COLUMN, NUMERICAL.DATA, ALPHA.NUMERICAL.DATA, DEFINE.SUBSCRIPTS, DATA.INPUT, SINGLE.READ, and ARRAY.READ), which are actually part of the language library as opposed to being part of the compiler. These procedures perform a compiler-like function, however, in that during the running of the ALGOL program assembled from HYDRO code, they read the HYDRO data cards, stored temporarily near the end of the SCRATCH AREA and later moved forward by the compiler to the end of the ALGOL program assembled there.

This Chapter should only be read after a working knowledge of the entire HYDRO system has been obtained from Chapters 2 and 3. The flow chart of this Chapter should be studied in conjunction with the program listings of the data read procedures given in Appendix III. All the format conventions for flow charts, listed at the beginning of Chapters 3 and 5, also apply to the flow chart in this Chapter.

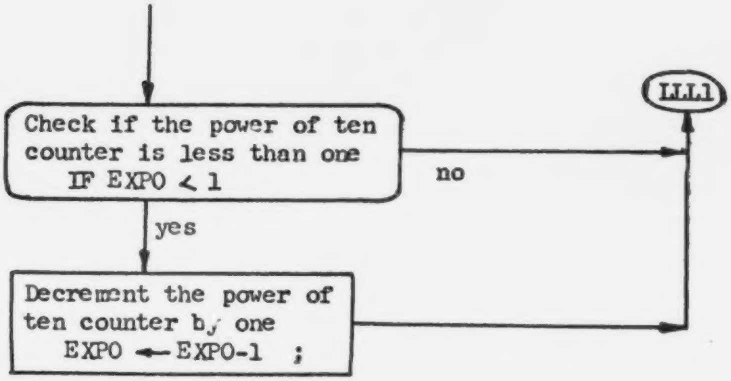
7.1 Flow Chart

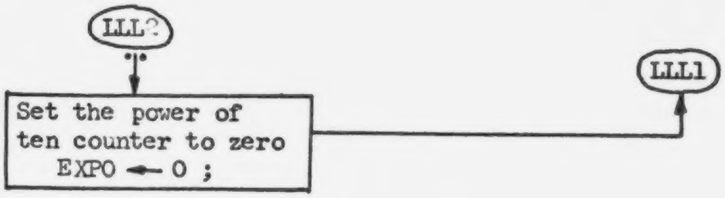

Declare the global variables for the data read procedures
the da II, JJ
SUB1, SUB2, SUB3
COLM, EXPO, TEMP, ABCD
NEGATIVE, ASTERISK, DEFINED, FINISHED
CHAR [1:80]
ALPH [1:80]
Declare NEXT.COLUMN
Flow Chart for NEXT.COLUMN
Enter NEXT.COLUMN
Increment the counter on CHAR by one
COLM ← COLM + 1
Check if the end of the data card image has been reached yet
COLM > 80
yes
no
Read alphanumerically the contents of the 80 card columns of the next data card into CHAR by NAME and READ statements
Reset the card column pointer to one
COLM ← 1
Leave NEXT.COLUMN

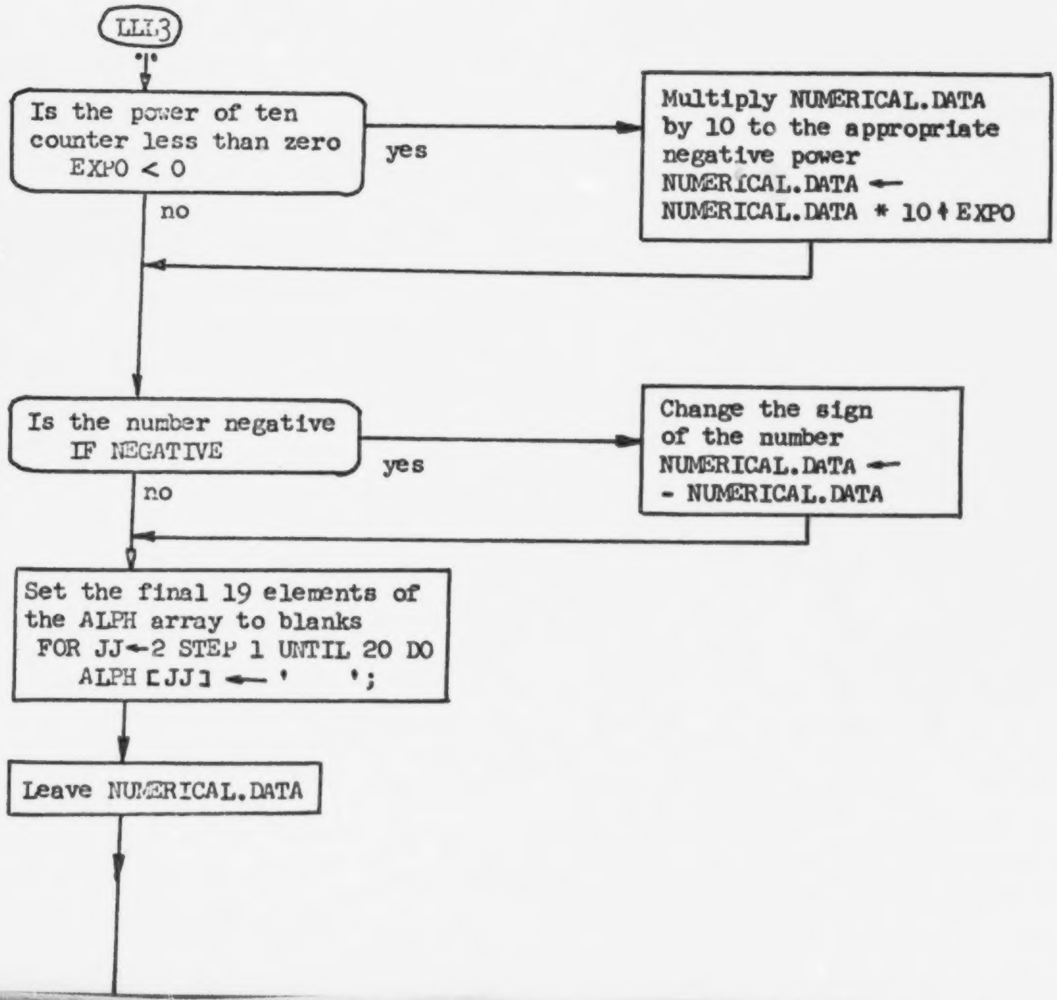

Declare NUMERICAL.DATA
Flow Chart for NUMERICAL.DATA
Enter NUMERICAL.DATA
Decrement the counter on the data card column to allow for the NEXT.COLUMN at LLL1
COLM ← COLM-1
Indicate that a negative number has not been found yet
NEGATIVE ← FALSE
Set the value of the procedure to zero
NUMERICAL.DATA ← 0
Set the power of ten counter, EXPO, to one
EXPO ← 1
LLL1
Call NEXT.COLUMN to increment the column counter or move to the next card
See Flow Chart for NEXT. COLUMN on Page 7.1.1

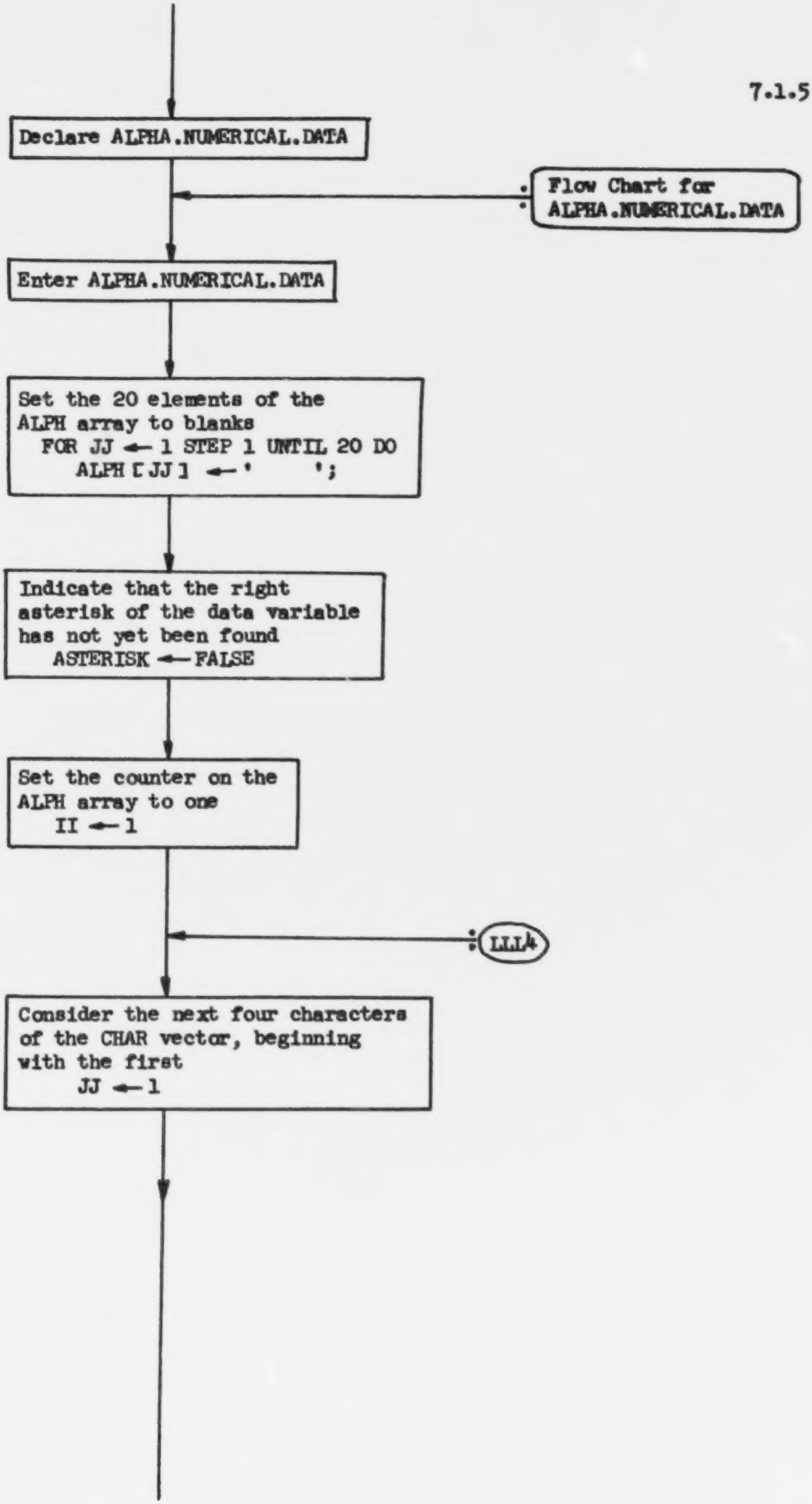

Does this column of CHAR contain a blank
IF CHAR [COLM] = ' '
yes
LLL1
no
Does this column of CHAR contain a plus
IF CHAR [COLM] = '+'
yes
LLL1
no
Does this column of CHAR contain a minus
IF CHAR [COLM] = '-'
yes
NEGATIVE ← TRUE
LLL1
no
Is CHAR [COLM] equal to one of the digits
IF CHAR [COLM] = '0'
IF CHAR [COLM] = '1'
IF CHAR [COLM] = '2'
IF CHAR [COLM] = '3'
IF CHAR [COLM] = '4'
IF CHAR [COLM] = '5'
IF CHAR [COLM] = '6'
IF CHAR [COLM] = '7'
IF CHAR [COLM] = '8'
IF CHAR [COLM] = '9'
yes
Store that digit into TEMP
THEN TEMP ← 0
THEN TEMP ← 1
THEN TEMP ← 2
THEN TEMP ← 3
THEN TEMP ← 4
THEN TEMP ← 5
THEN TEMP ← 6
THEN TEMP ← 7
THEN TEMP ← 8
THEN TEMP ← 9
no
Does CHAR [COLM] contain a period
IF CHAR [COLM] = '.'
yes
LLL2
no
Is CHAR [COLM] equal to a comma, semicolon, or right bracket
IF CHAR [COLM] = ',' ∨
CHAR [COLM] = ';' ∨
CHAR [COLM] = ']'
yes
LLL3
no
Set the value of the procedure to ten times what it was before, plus TEMP
NUMERICAL.DATA ←
10 * NUMERICAL.DATA + TEMP ;

7.1.4

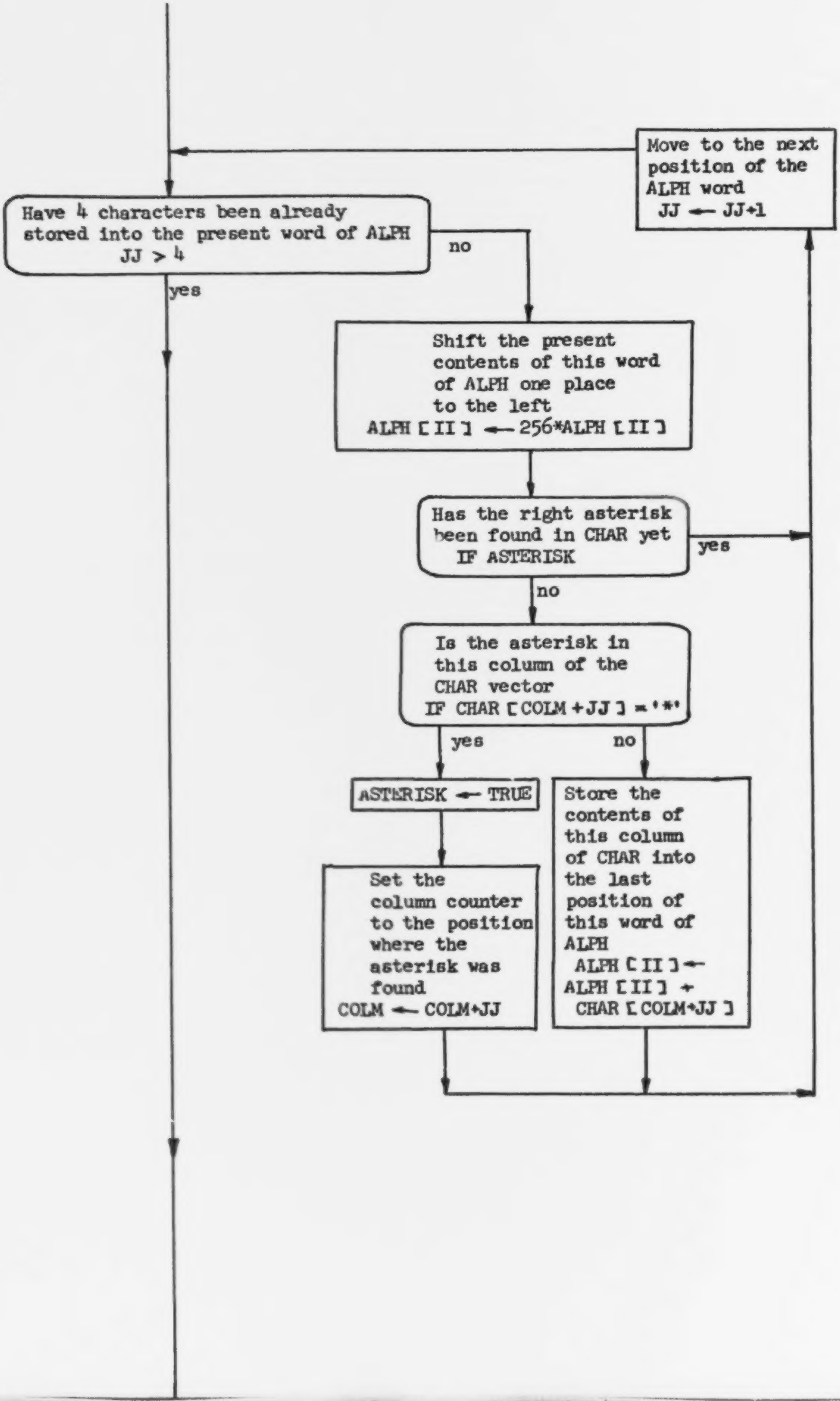

Check if the power of ten counter is less than one
IF EXPO < 1
no
LLL1
yes
Decrement the power of ten counter by one
EXPO ← EXPO-1 ;

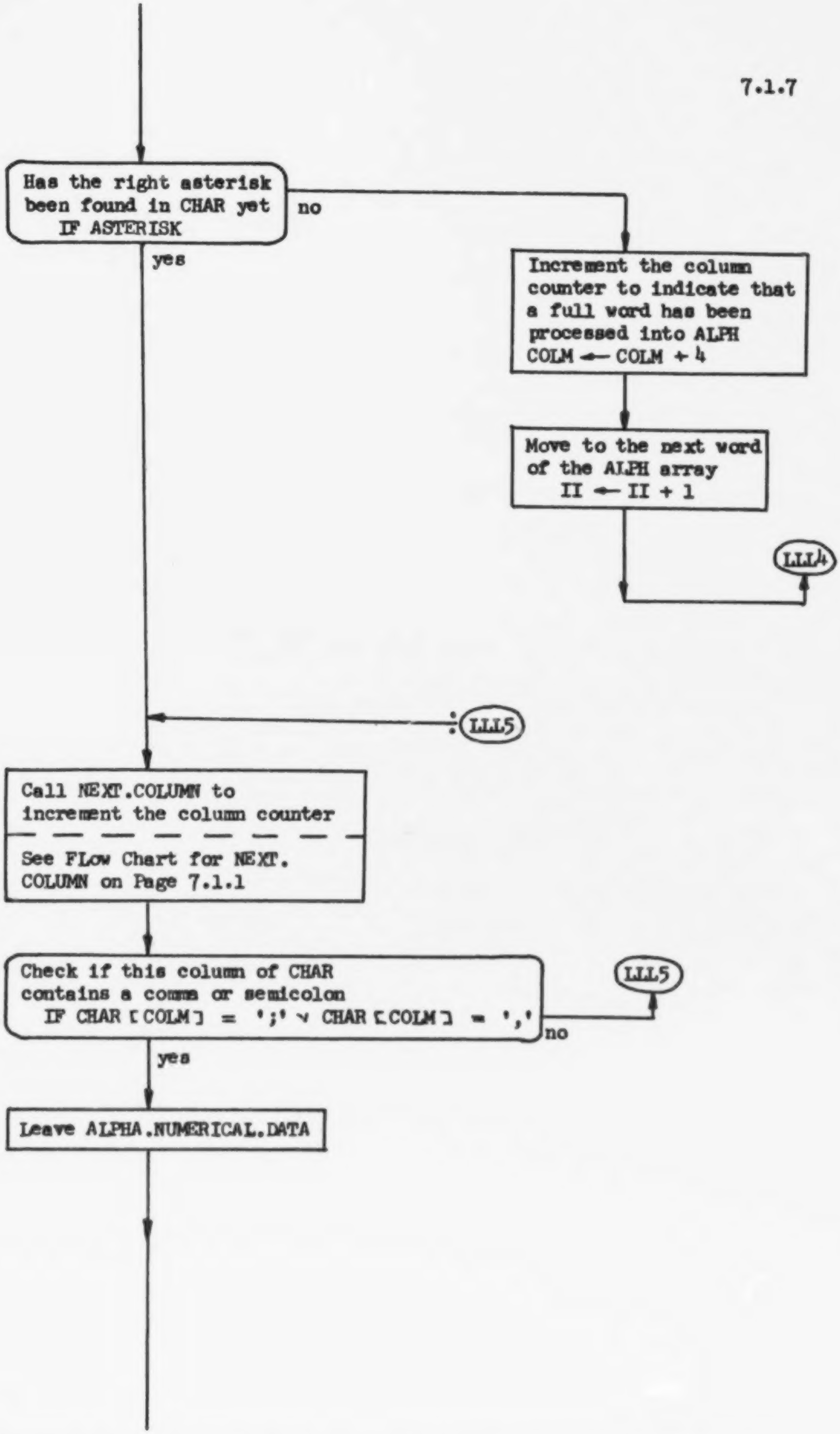

LLL2
Set the power of ten counter to zero
EXPO ← 0 ;
LLL1

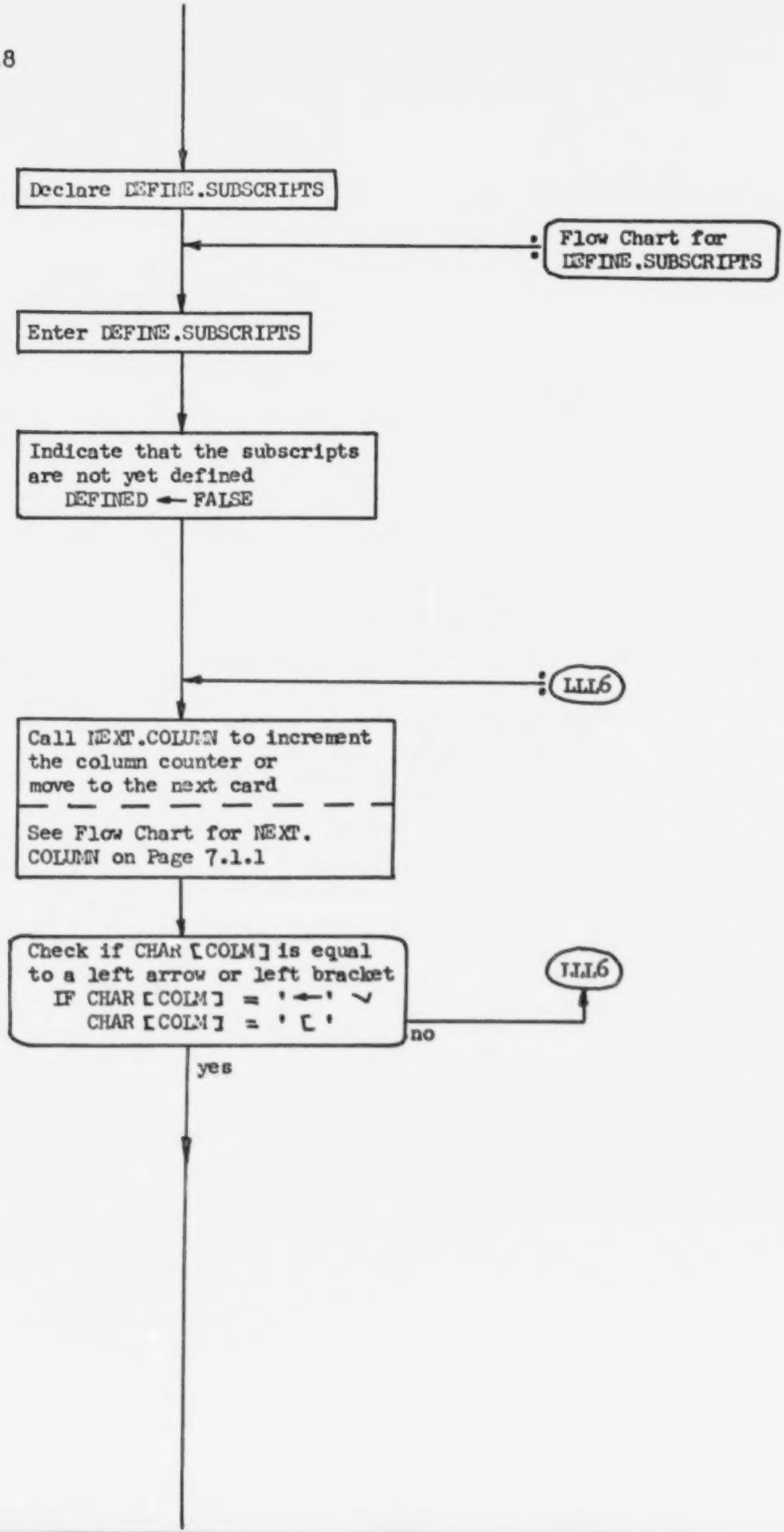

LLL3
Is the power of ten counter less than zero
EXPO < 0
yes
Multiply NUMERICAL.DATA by 10 to the appropriate negative power
NUMERICAL.DATA ←
NUMERICAL.DATA * 10 ↑ EXPO
no
Is the number negative
IF NEGATIVE
yes
Change the sign of the number
NUMERICAL.DATA ←
- NUMERICAL.DATA
no
Set the final 19 elements of the ALPH array to blanks
FOR JJ←2 STEP 1 UNTIL 20 DO
ALPH [JJ] ← '   ';
Leave NUMERICAL.DATA

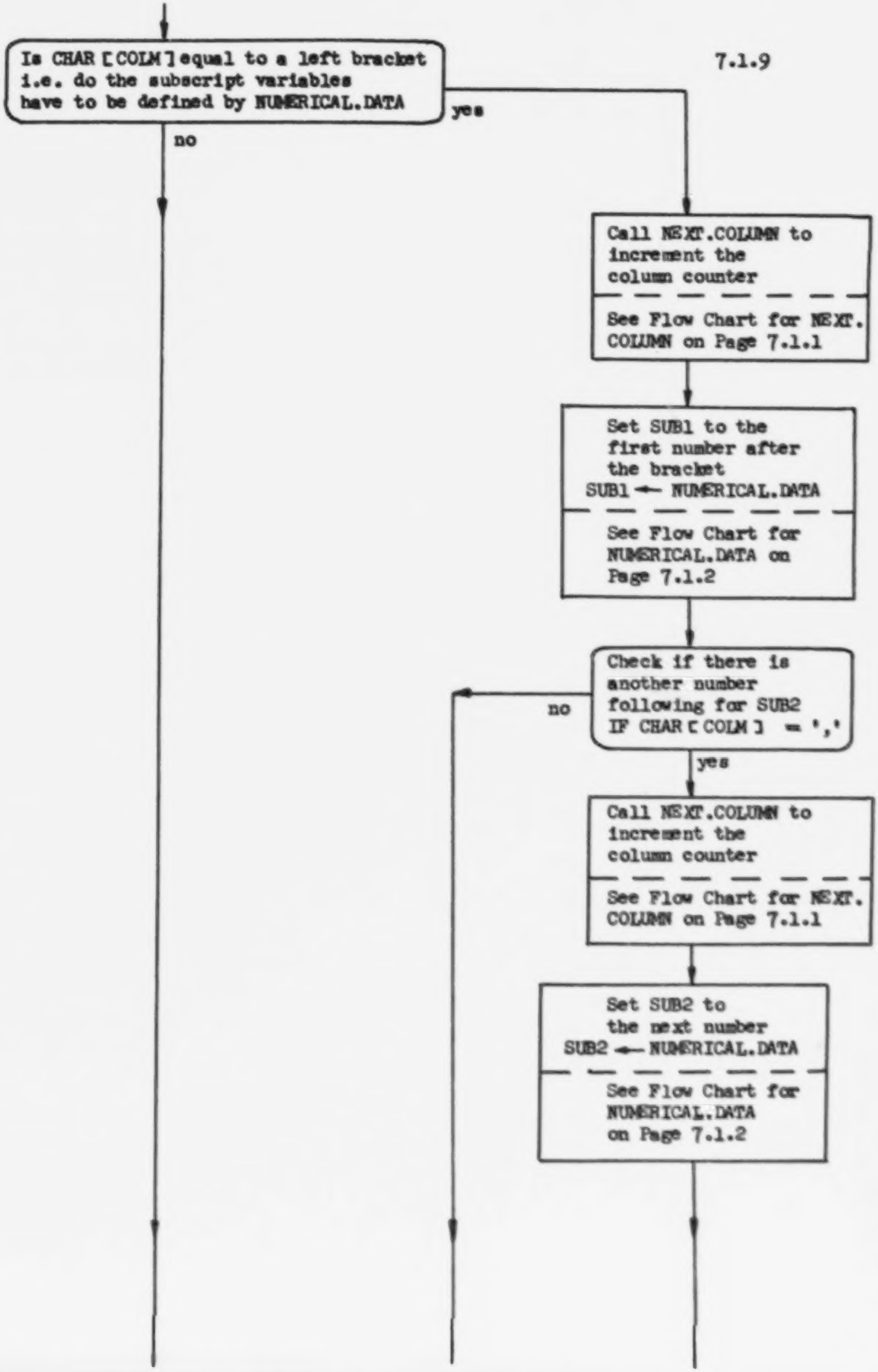

Declare ALPHA.NUMERICAL.DATA
Flow Chart for ALPHA.NUMERICAL.DATA
Enter ALPHA.NUMERICAL.DATA
Set the 20 elements of the ALPH array to blanks
FOR JJ ← 1 STEP 1 UNTIL 20 DO
ALPH [JJ] ← '   ';
Indicate that the right asterisk of the data variable has not yet been found
ASTERISK ← FALSE
Set the counter on the ALPH array to one
II ← 1
LL14
Consider the next four characters of the CHAR vector, beginning with the first
JJ ← 1

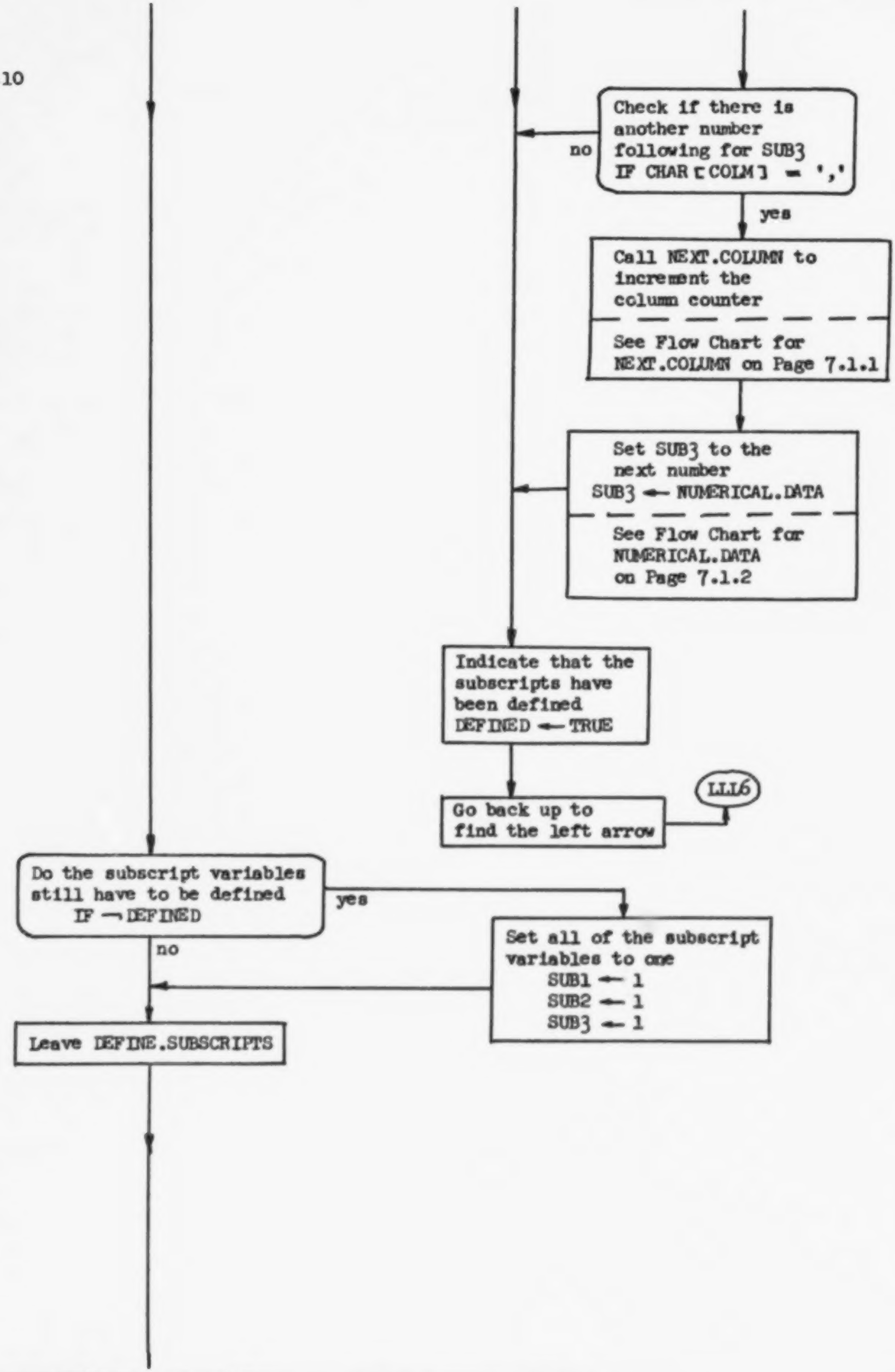
7.1.6

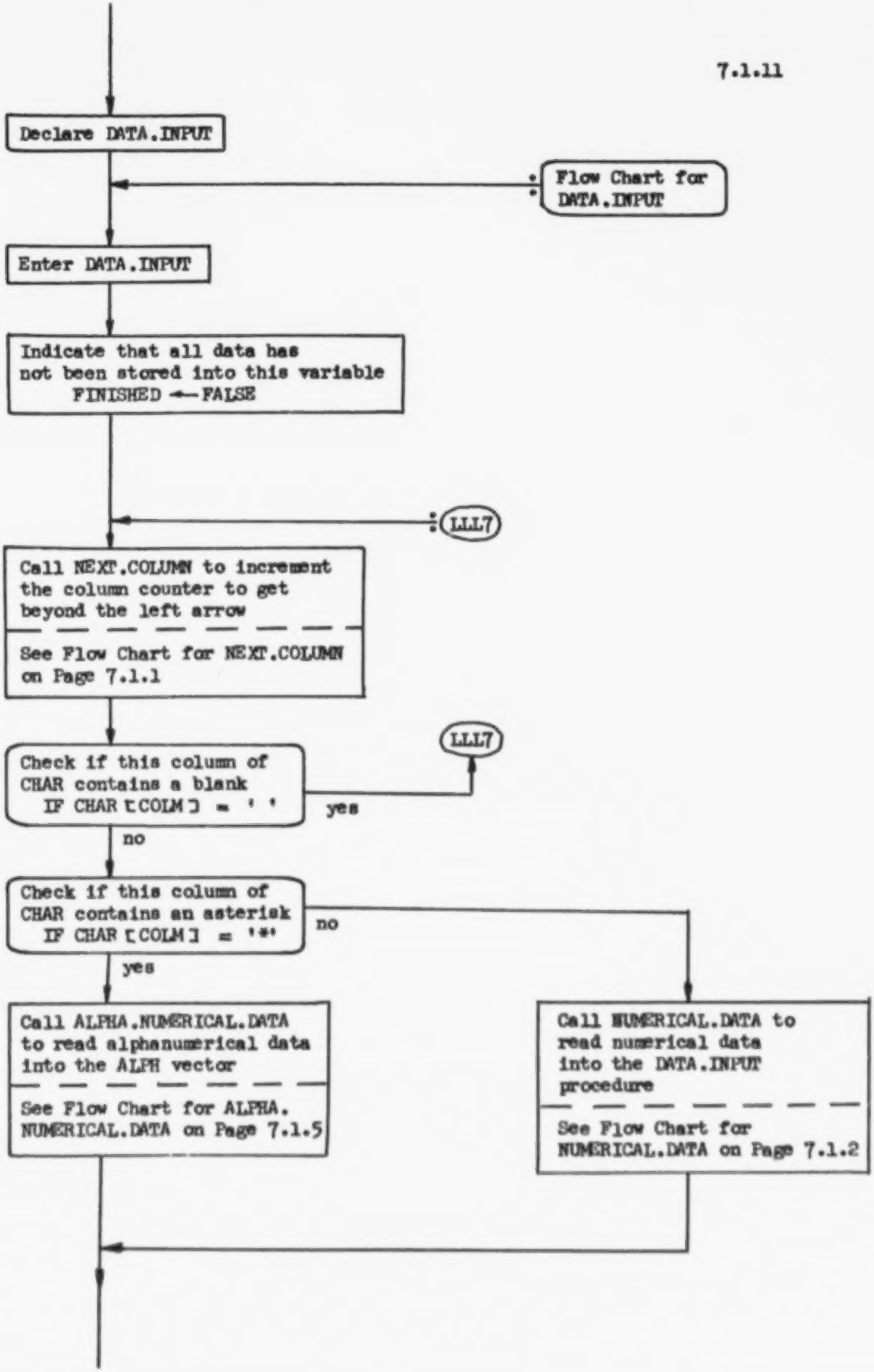

Have 4 characters been already stored into the present word of ALPH
JJ > 4
no
yes
Move to the next position of the ALPH word
JJ ← JJ+1
Shift the present contents of this word of ALPH one place to the left
ALPH [II] ← 256*ALPH [II]
Has the right asterisk been found in CHAR yet
IF ASTERISK
yes
no
Is the asterisk in this column of the CHAR vector
IF CHAR [COLM + JJ] = '*'
yes
no
ASTERISK ← TRUE
Store the contents of this column of CHAR into the last position of this word of ALPH
ALPH [II] ← ALPH [II] + CHAR [COLM+JJ]
Set the column counter to the position where the asterisk was found
COLM ← COLM+JJ

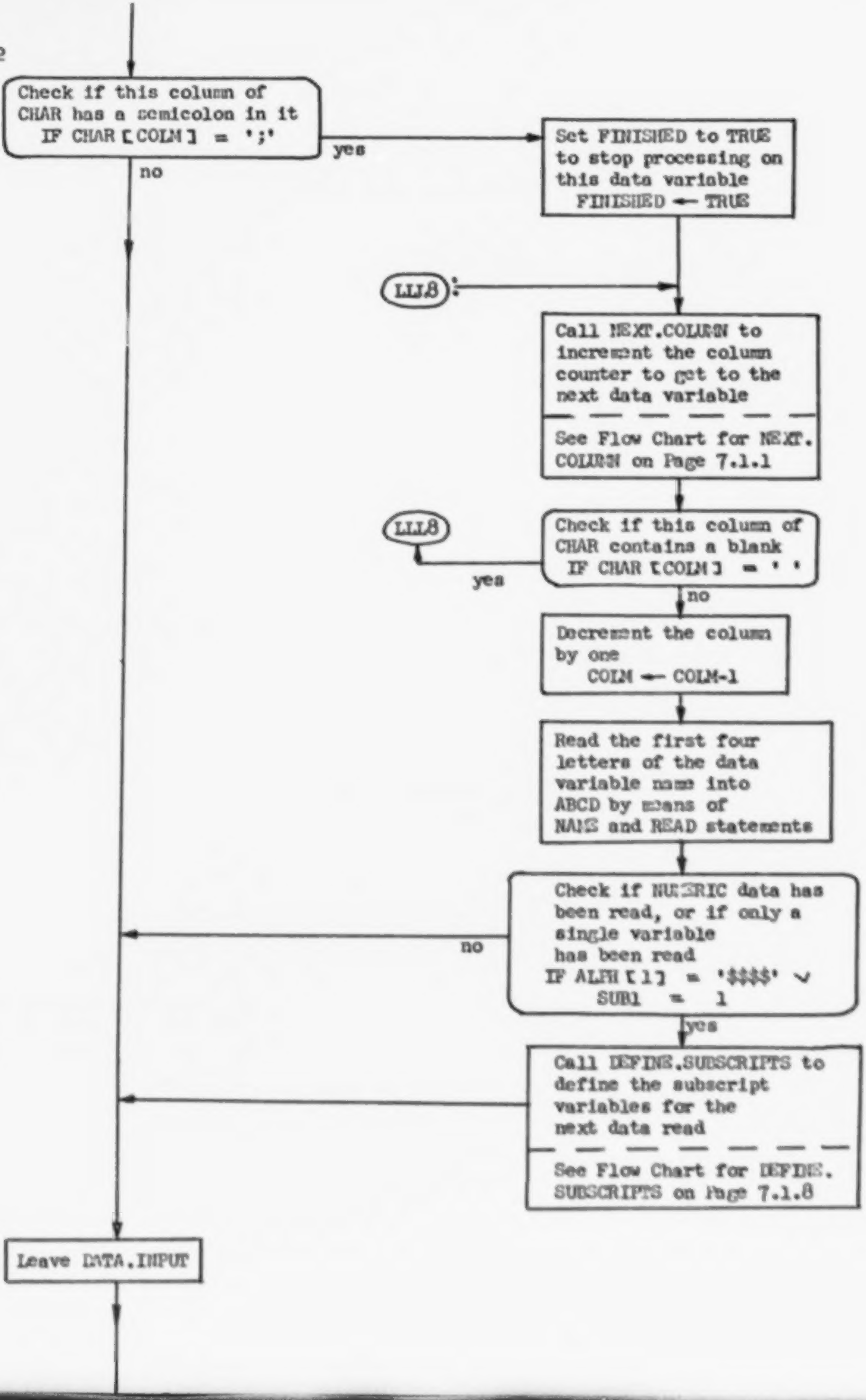

Has the right asterisk been found in CHAR yet
IF ASTERISK
no
yes
Increment the column counter to indicate that a full word has been processed into ALPH
COLM ← COLM + 4
Move to the next word of the ALPH array
II ← II + 1
LLL4
LLL5
Call NEXT.COLUMN to increment the column counter
See FLow Chart for NEXT. COLUMN on Page 7.1.1
Check if this column of CHAR contains a comma or semicolon
IF CHAR [COLM] = ';' ∨ CHAR [COLM] = ','
no
LLL5
yes
Leave ALPHA.NUMERICAL.DATA

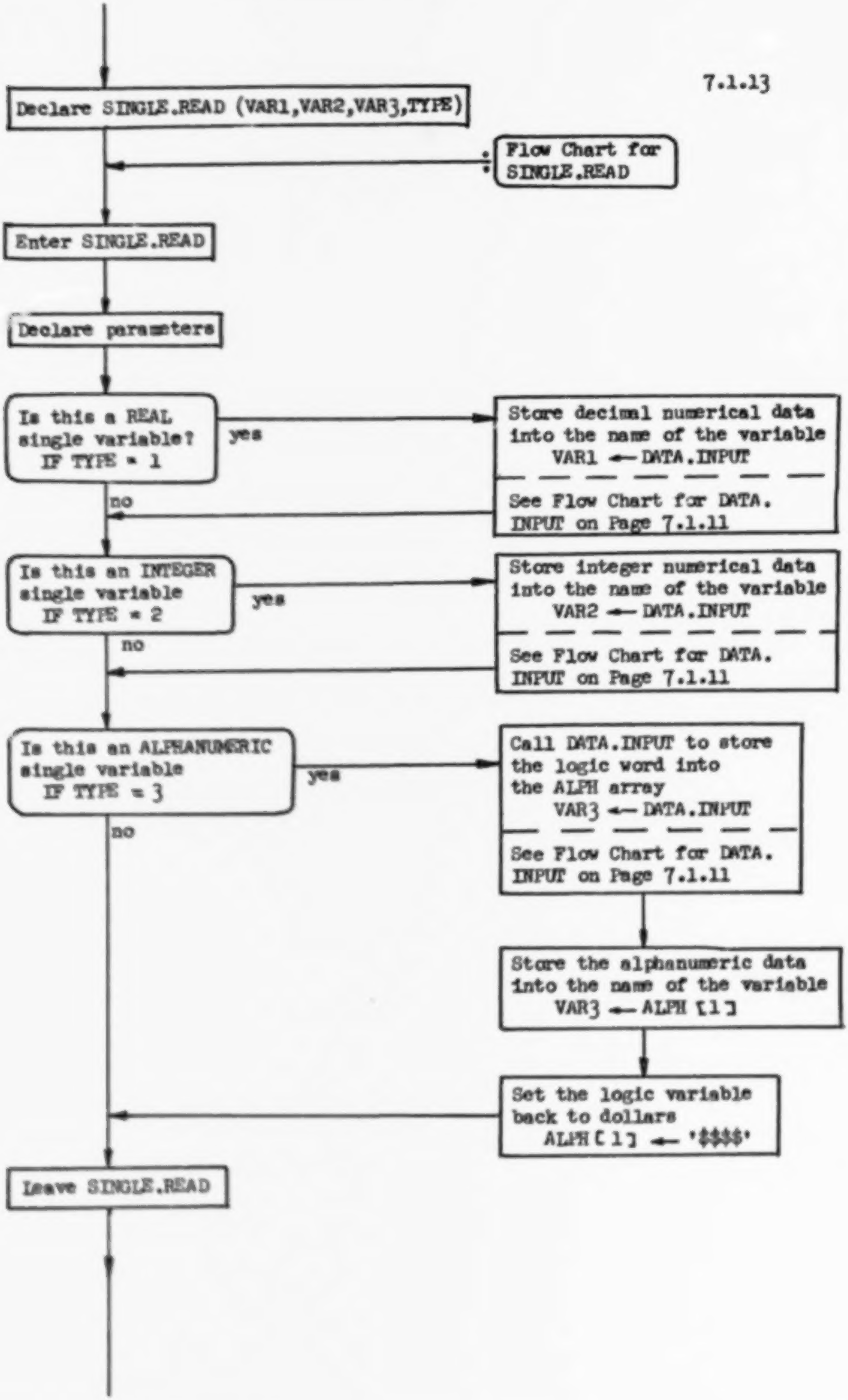

Declare DEFINE.SUBSCRIPTS
Flow Chart for DEFINE.SUBSCRIPTS
Enter DEFINE.SUBSCRIPTS
Indicate that the subscripts are not yet defined
DEFINED ← FALSE
LLL6
Call NEXT.COLUMN to increment the column counter or move to the next card
See Flow Chart for NEXT. COLUMN on Page 7.1.1
Check if CHAR [COLM] is equal to a left arrow or left bracket
IF CHAR [COLM] = '←' ∨
CHAR [COLM] = '['
no
LLL6
yes

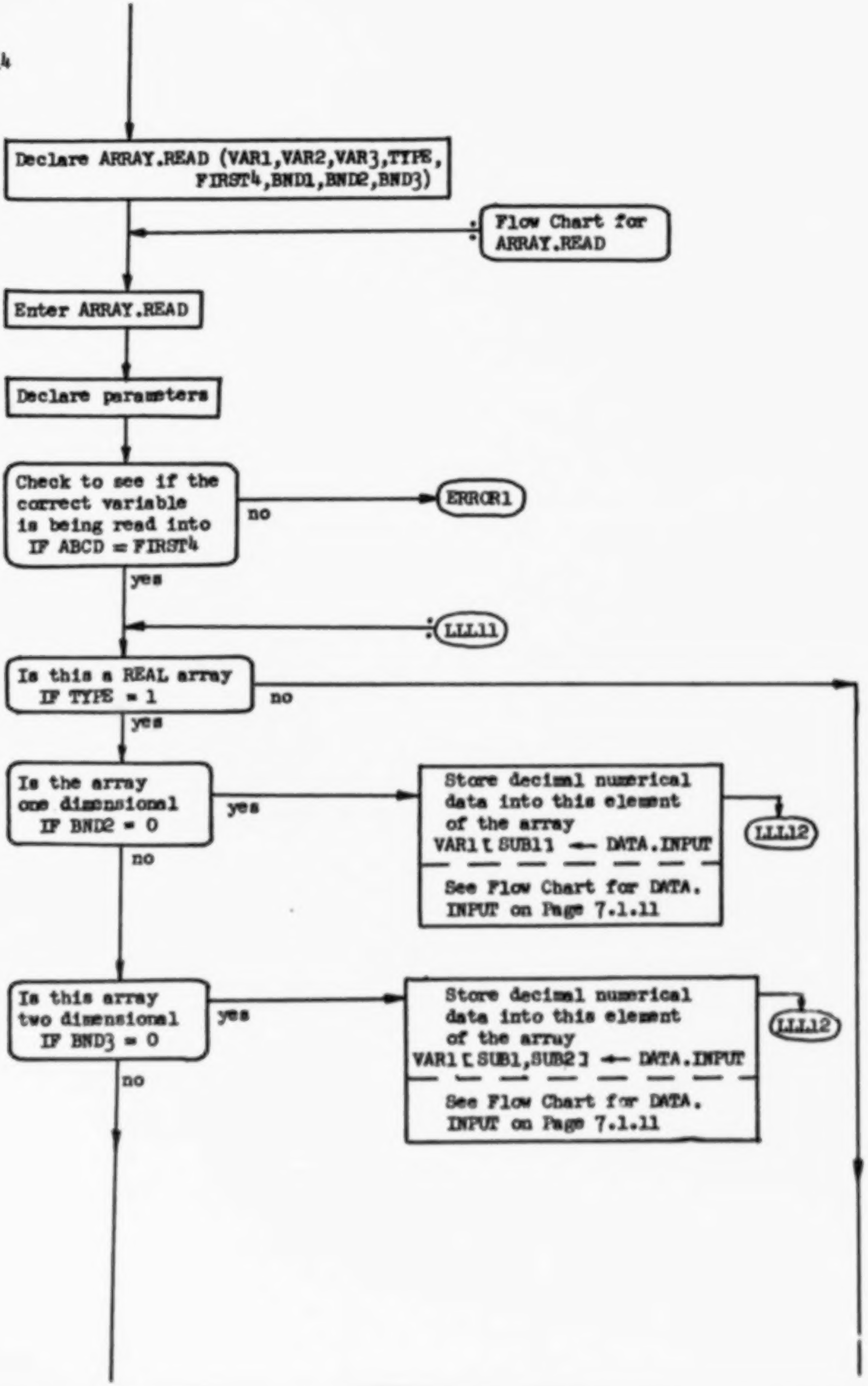

Is CHAR [COLM] equal to a left bracket i.e. do the subscript variables have to be defined by NUMERICAL.DATA
yes
no
Call NEXT.COLUMN to increment the column counter
See Flow Chart for NEXT. COLUMN on Page 7.1.1
Set SUB1 to the first number after the bracket
SUB1 ← NUMERICAL.DATA
See Flow Chart for NUMERICAL.DATA on Page 7.1.2
Check if there is another number following for SUB2
IF CHAR [COLM] = ','
no
yes
Call NEXT.COLUMN to increment the column counter
See Flow Chart for NEXT. COLUMN on Page 7.1.1
Set SUB2 to the next number
SUB2 ← NUMERICAL.DATA
See Flow Chart for NUMERICAL.DATA on Page 7.1.2

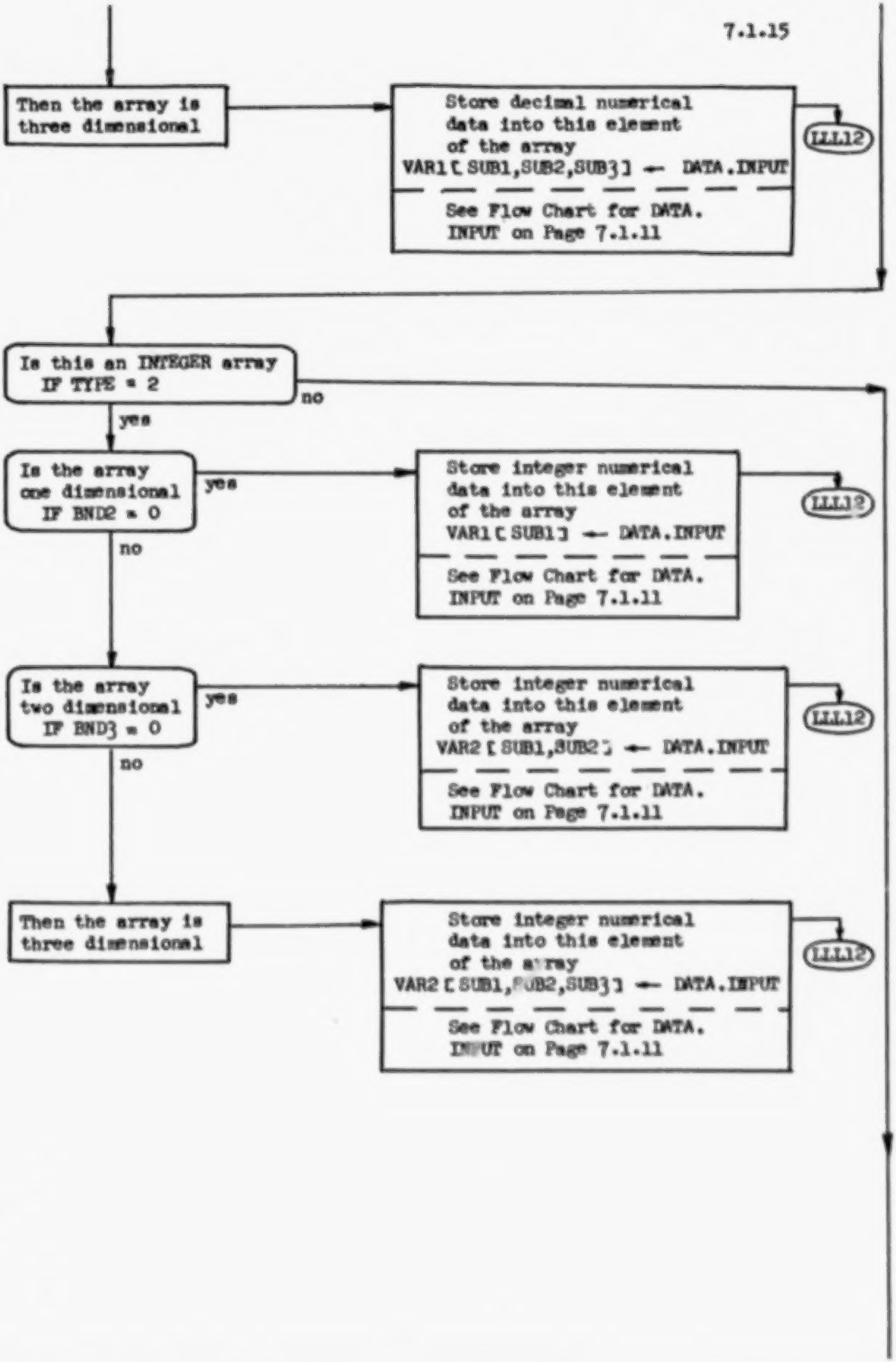

Check if there is another number following for SUB3 IF CHAR [COLM] = ','
no
yes
Call NEXT.COLUMN to increment the column counter
See Flow Chart for NEXT.COLUMN on Page 7.1.1
Set SUB3 to the next number SUB3 ← NUMERICAL.DATA
See Flow Chart for NUMERICAL.DATA on Page 7.1.2
Indicate that the subscripts have been defined DEFINED ← TRUE
Go back up to find the left arrow
LL16
Do the subscript variables still have to be defined IF ¬ DEFINED
yes
no
Set all of the subscript variables to one
SUB1 ← 1
SUB2 ← 1
SUB3 ← 1
Leave DEFINE.SUBSCRIPTS

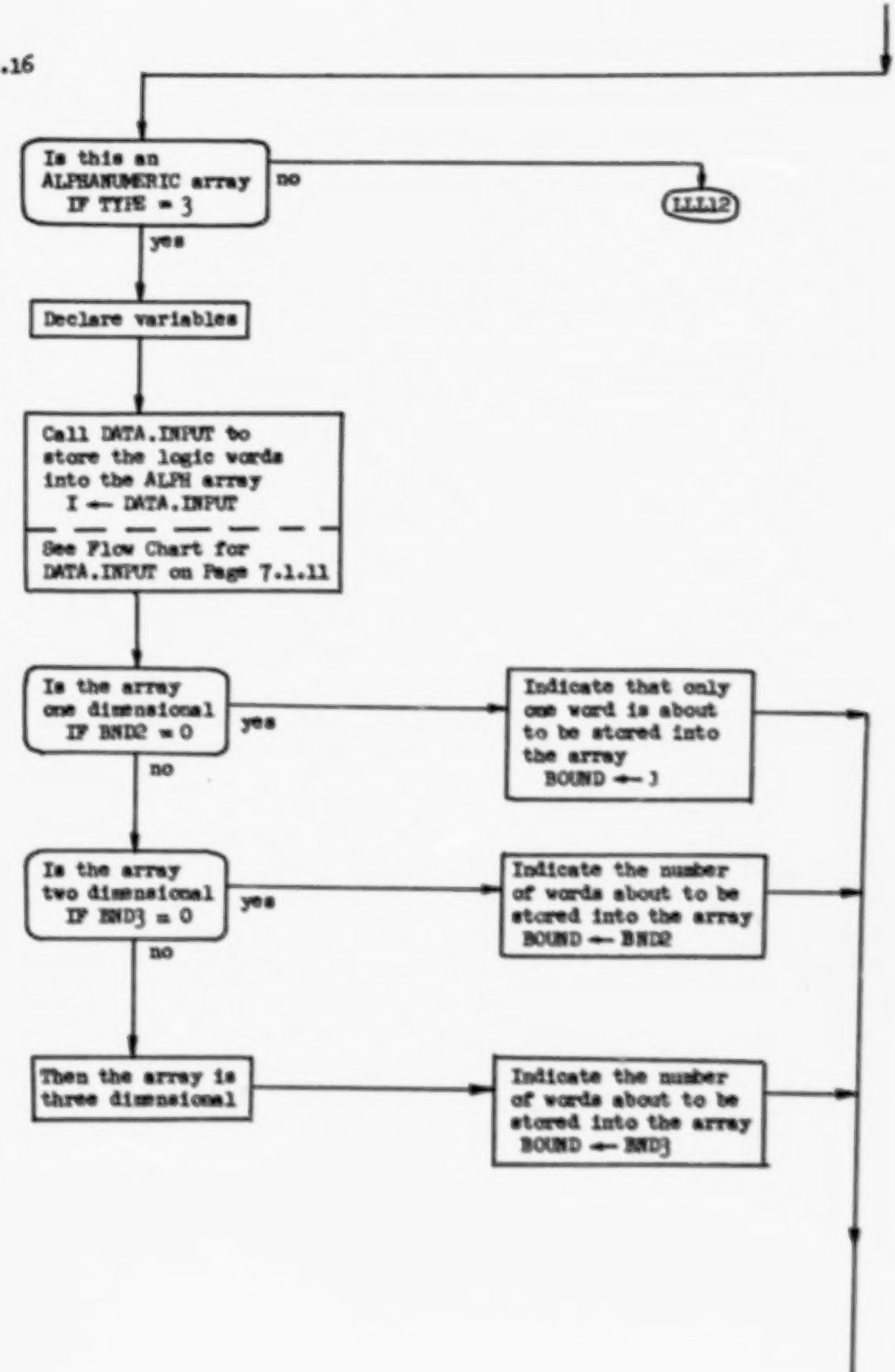

Declare DATA.INPUT
Flow Chart for DATA.INPUT
Enter DATA.INPUT
Indicate that all data has not been stored into this variable
FINISHED ← FALSE
LLL7
Call NEXT.COLUMN to increment the column counter to get beyond the left arrow
See Flow Chart for NEXT.COLUMN on Page 7.1.1
Check if this column of CHAR contains a blank
IF CHAR [COLM] = ' '
yes
LLL7
no
Check if this column of CHAR contains an asterisk
IF CHAR [COLM] = '*'
no
yes
Call ALPHA.NUMERICAL.DATA to read alphanumerical data into the ALPH vector
See Flow Chart for ALPHA. NUMERICAL.DATA on Page 7.1.5
Call NUMERICAL.DATA to read numerical data into the DATA.INPUT procedure
See Flow Chart for NUMERICAL.DATA on Page 7.1.2

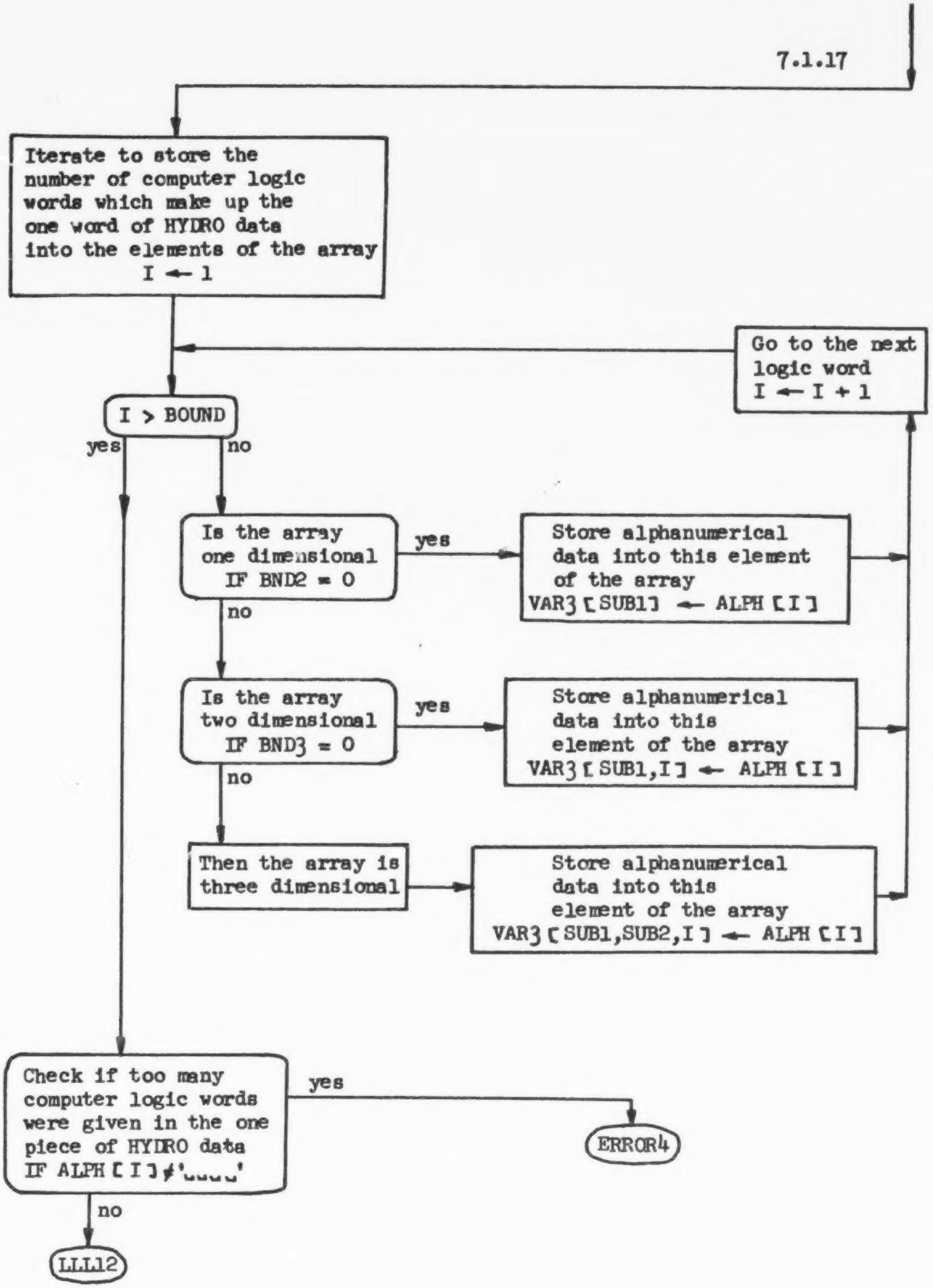

Check if this column of CHAR has a semicolon in it
IF CHAR [COLM] = ';'
yes
no
Set FINISHED to TRUE to stop processing on this data variable
FINISHED ← TRUE
LLL8
Call NEXT.COLUMN to increment the column counter to get to the next data variable
See Flow Chart for NEXT. COLUMN on Page 7.1.1
LLL8
Check if this column of CHAR contains a blank
IF CHAR [COLM] = ' '
yes
no
Decrement the column by one
COLM ← COLM-1
Read the first four letters of the data variable name into ABCD by means of NAME and READ statements
Check if NUMERIC data has been read, or if only a single variable has been read
IF ALPH [1] = '$$$$' ∨
SUB1 = 1
no
yes
Call DEFINE.SUBSCRIPTS to define the subscript variables for the next data read
See Flow Chart for DEFINE. SUBSCRIPTS on Page 7.1.8
Leave DATA.INPUT

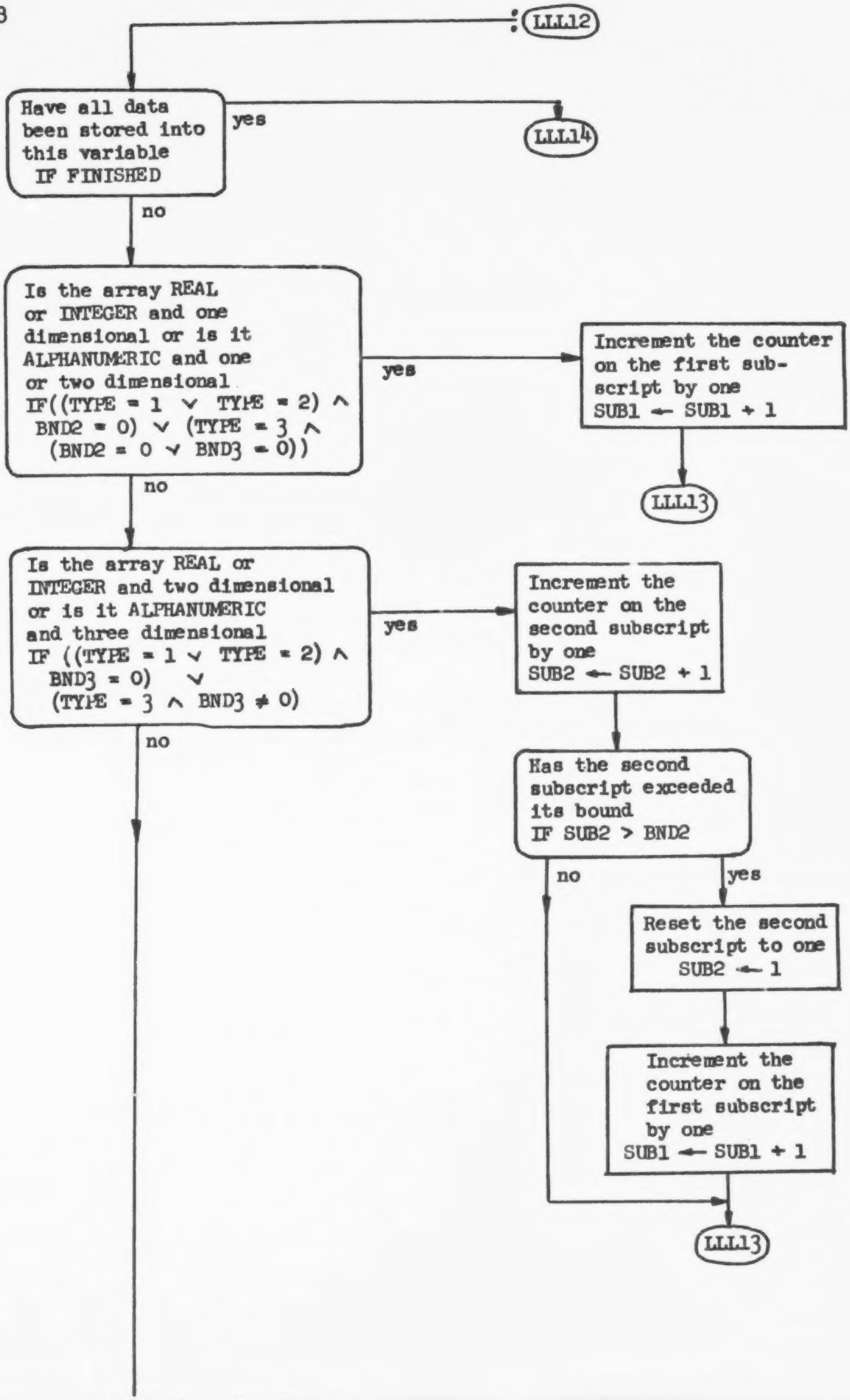

Declare SINGLE.READ (VAR1,VAR2,VAR3,TYPE)
Flow Chart for SINGLE.READ
Enter SINGLE.READ
Declare parameters
Is this a REAL single variable? IF TYPE = 1
yes
Store decimal numerical data into the name of the variable VAR1 ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
no
Is this an INTEGER single variable IF TYPE = 2
yes
Store integer numerical data into the name of the variable VAR2 ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
no
Is this an ALPHANUMERIC single variable IF TYPE = 3
yes
Call DATA.INPUT to store the logic word into the ALPH array VAR3 ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
no
Store the alphanumeric data into the name of the variable VAR3 ← ALPH [1]
Set the logic variable back to dollars ALPH [1] ← '$$$$'
Leave SINGLE.READ

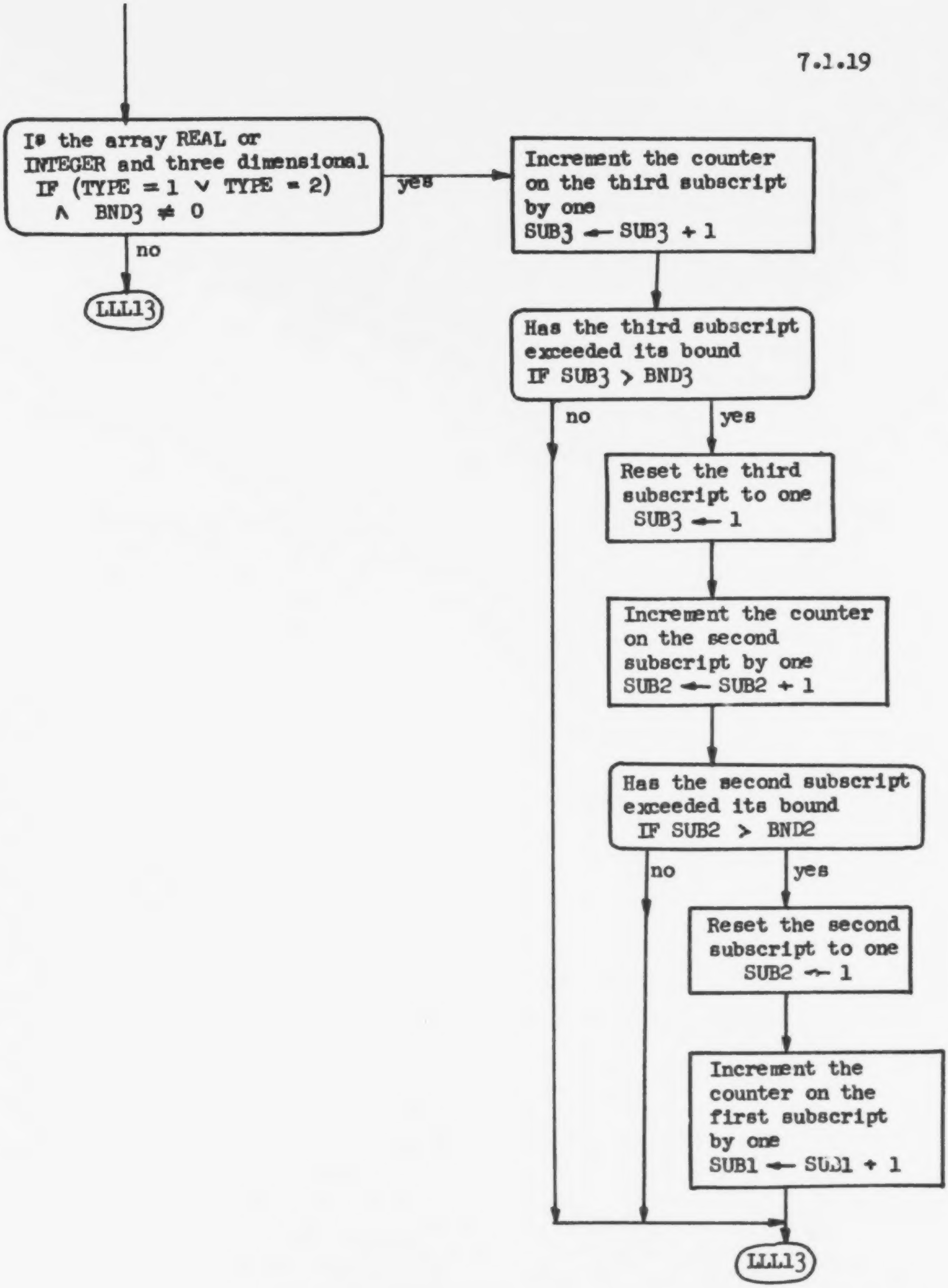

Declare ARRAY.READ (VAR1,VAR2,VAR3,TYPE, FIRST4,BND1,BND2,BND3)
Flow Chart for ARRAY.READ
Enter ARRAY.READ
Declare parameters
Check to see if the correct variable is being read into IF ABCD = FIRST4
no
ERROR1
yes
LLL11
Is this a REAL array IF TYPE = 1
no
yes
Is the array one dimensional IF BND2 = 0
yes
Store decimal numerical data into this element of the array VAR1[SUB1] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12
no
Is this array two dimensional IF BND3 = 0
yes
Store decimal numerical data into this element of the array VAR1[SUB1,SUB2] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12
no

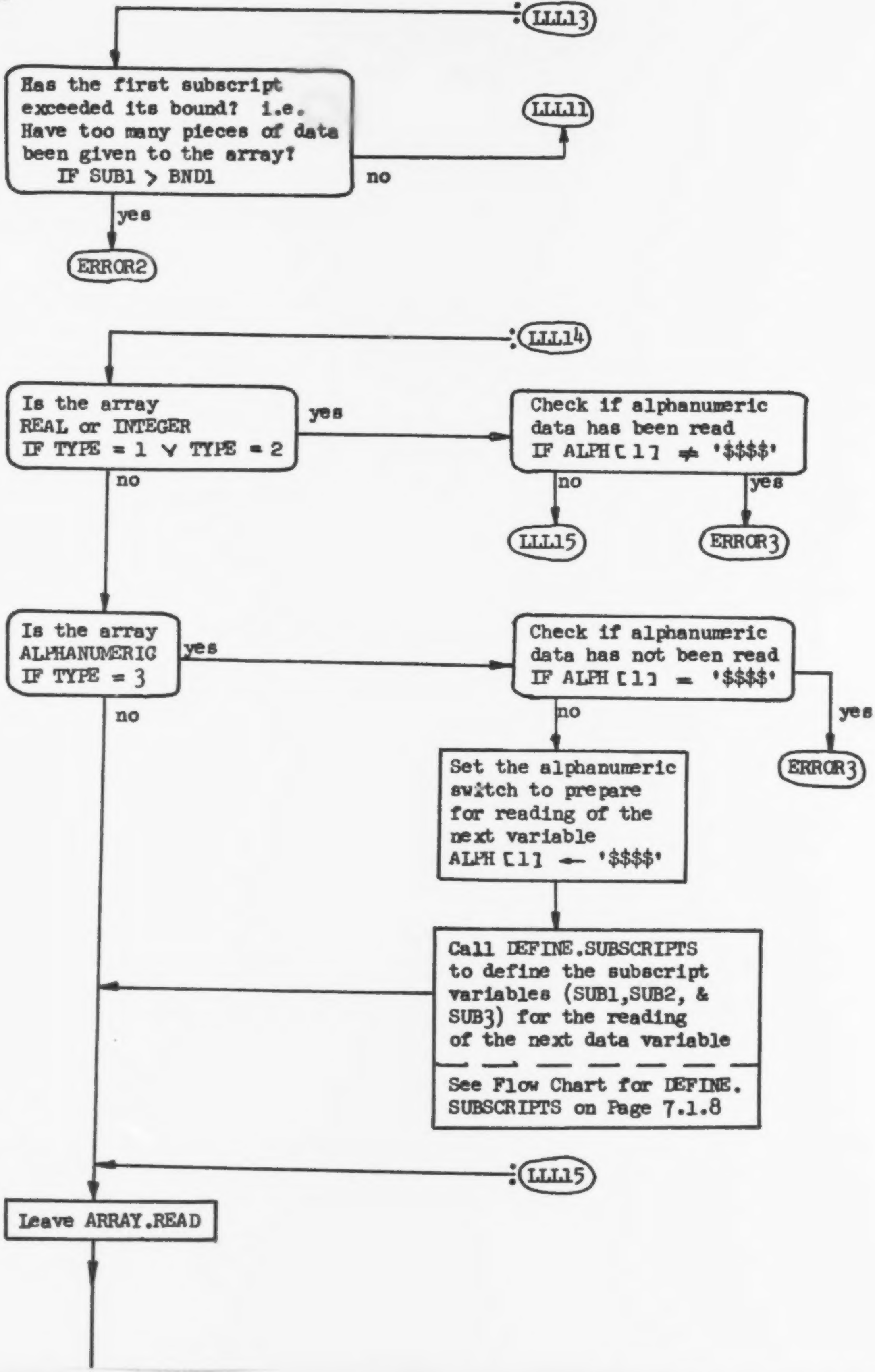

Then the array is three dimensional
Store decimal numerical data into this element of the array
VAR1 [SUB1,SUB2,SUB3] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12
Is this an INTEGER array IF TYPE = 2
no
yes
Is the array one dimensional IF BND2 = 0
yes
no
Store integer numerical data into this element of the array
VAR1 [SUB1] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12
Is the array two dimensional IF BND3 = 0
yes
no
Store integer numerical data into this element of the array
VAR2 [SUB1,SUB2] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12
Then the array is three dimensional
Store integer numerical data into this element of the array
VAR2 [SUB1,SUB2,SUB3] ← DATA.INPUT
See Flow Chart for DATA. INPUT on Page 7.1.11
LLL12

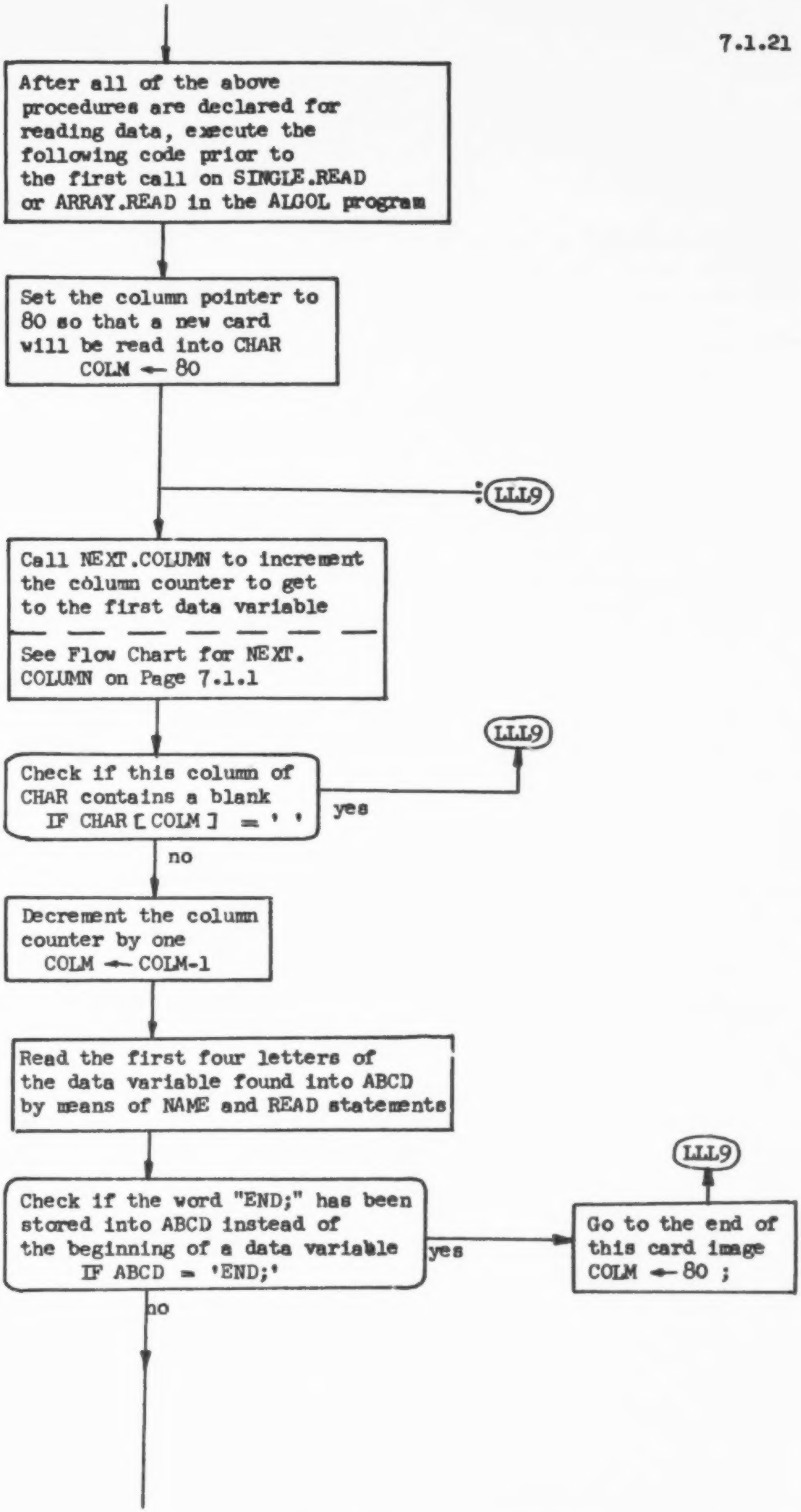

Is this an
ALPHANUMERIC array
IF TYPE = 3
no
LLL12
yes
Declare variables
Call DATA.INPUT to
store the logic words
into the ALPH array
I ← DATA.INPUT
See Flow Chart for
DATA.INPUT on Page 7.1.11
Is the array
one dimensional
IF BND2 = 0
yes
Indicate that only
one word is about
to be stored into
the array
BOUND ← 1
no
Is the array
two dimensional
IF BND3 = 0
yes
Indicate the number
of words about to be
stored into the array
BOUND ← BND2
no
Then the array is
three dimensional
Indicate the number
of words about to be
stored into the array
BOUND ← BND3

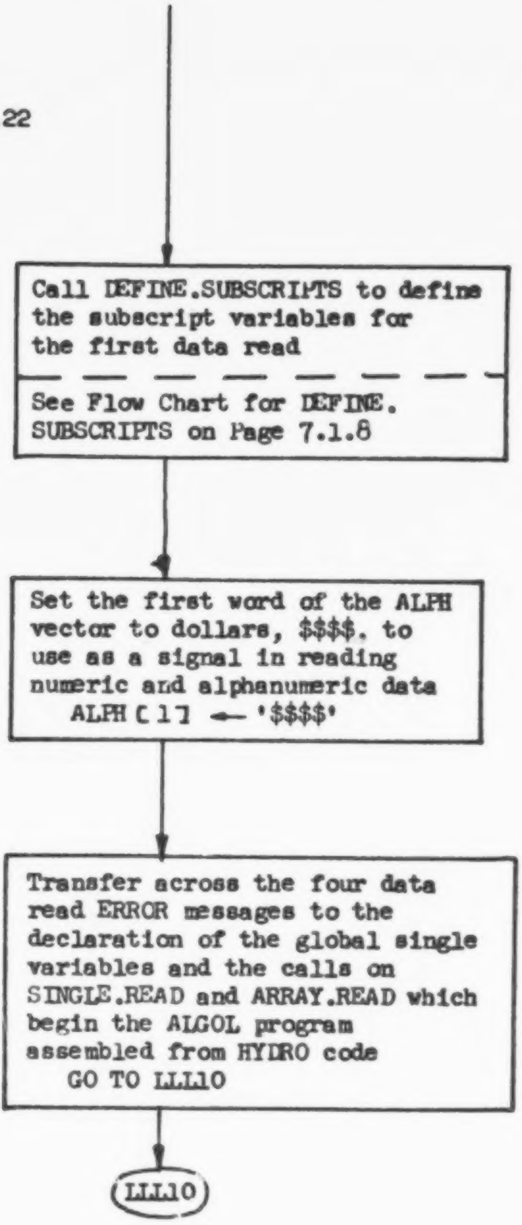

Iterate to store the number of computer logic words which make up the one word of HYDRO data into the elements of the array I ← 1
Go to the next logic word I ← I + 1
I > BOUND
yes
no
Is the array one dimensional IF BND2 = 0
yes
Store alphanumerical data into this element of the array VAR3 [SUB1] ← ALPH [I]
no
Is the array two dimensional IF BND3 = 0
yes
Store alphanumerical data into this element of the array VAR3 [SUB1,I] ← ALPH [I]
no
Then the array is three dimensional
Store alphanumerical data into this element of the array VAR3 [SUB1,SUB2,I] ← ALPH [I]
Check if too many computer logic words were given in the one piece of HYDRO data IF ALPH [I] ≠ '␣␣␣␣'
yes
ERROR4
no
LLL12


LLL12
Have all data been stored into this variable
IF FINISHED
yes
LLL14
no
Is the array REAL or INTEGER and one dimensional or is it ALPHANUMERIC and one or two dimensional
IF((TYPE = 1 ∨ TYPE = 2) ∧ BND2 = 0) ∨ (TYPE = 3 ∧ (BND2 = 0 ∨ BND3 = 0))
yes
Increment the counter on the first subscript by one
SUB1 ← SUB1 + 1
LLL13
no
Is the array REAL or INTEGER and two dimensional or is it ALPHANUMERIC and three dimensional
IF ((TYPE = 1 ∨ TYPE = 2) ∧ BND3 = 0) ∨ (TYPE = 3 ∧ BND3 ≠ 0)
yes
Increment the counter on the second subscript by one
SUB2 ← SUB2 + 1
no
Has the second subscript exceeded its bound
IF SUB2 > BND2
no
yes
Reset the second subscript to one
SUB2 ← 1
Increment the counter on the first subscript by one
SUB1 ← SUB1 + 1
LLL13


Is the array REAL or INTEGER and three dimensional
IF (TYPE = 1 ∨ TYPE = 2) ∧ BND3 ≠ 0
yes
no
LLL13
Increment the counter on the third subscript by one
SUB3 ← SUB3 + 1
Has the third subscript exceeded its bound
IF SUB3 > BND3
no
yes
Reset the third subscript to one
SUB3 ← 1
Increment the counter on the second subscript by one
SUB2 ← SUB2 + 1
Has the second subscript exceeded its bound
IF SUB2 > BND2
no
yes
Reset the second subscript to one
SUB2 ← 1
Increment the counter on the first subscript by one
SUB1 ← SUB1 + 1
LLL13


LLL13
Has the first subscript exceeded its bound? i.e. Have too many pieces of data been given to the array? IF SUB1 > BND1
no
LLL11
yes
ERROR2
LLL14
Is the array REAL or INTEGER IF TYPE = 1 ∨ TYPE = 2
yes
Check if alphanumeric data has been read IF ALPH [1] ≠ '$$$$'
no
LLL15
yes
ERROR3
no
Is the array ALPHANUMERIC IF TYPE = 3
yes
Check if alphanumeric data has not been read IF ALPH [1] = '$$$$'
no
yes
ERROR3
no
Set the alphanumeric switch to prepare for reading of the next variable ALPH [1] ← '$$$$'
Call DEFINE.SUBSCRIPTS to define the subscript variables (SUB1,SUB2, & SUB3) for the reading of the next data variable
See Flow Chart for DEFINE. SUBSCRIPTS on Page 7.1.8
LLL15
Leave ARRAY.READ


After all of the above procedures are declared for reading data, execute the following code prior to the first call on SINGLE.READ or ARRAY.READ in the ALGOL program
Set the column pointer to 80 so that a new card will be read into CHAR COLM ← 80
LLL9
Call NEXT.COLUMN to increment the column counter to get to the first data variable
See Flow Chart for NEXT. COLUMN on Page 7.1.1
LLL9
Check if this column of CHAR contains a blank IF CHAR [COLM] = ' '
yes
no
Decrement the column counter by one COLM ← COLM-1
Read the first four letters of the data variable found into ABCD by means of NAME and READ statements
LLL9
Check if the word "END;" has been stored into ABCD instead of the beginning of a data variable IF ABCD = 'END;'
yes
Go to the end of this card image COLM ← 80 ;
no



Call DEFINE.SUBSCRIPTS to define the subscript variables for the first data read

See Flow Chart for DEFINE. SUBSCRIPTS on Page 7.1.8

Set the first word of the ALPH vector to dollars, $$$$. to use as a signal in reading numeric and alphanumeric data

ALPH [1] ← '$$$$'

Transfer across the four data read ERROR messages to the declaration of the global single variables and the calls on SINGLE.READ and ARRAY.READ which begin the ALGOL program assembled from HYDRO code

GO TO LLL10

7.2 Explanation of the Flow Chart

The principal part of this flow chart deals with the declaration of the seven data reading procedures - NEXT.COLUMN, NUMERICAL.DATA, ALPHA.NUMERICAL.DATA, DEFINE.SUBSCRIPTS, DATA.INPUT, SINGLE.READ, and ARRAY.READ. These procedures build upon each other, so that the latter procedures contain calls on those declared previously. The NEXT.COLUMN procedure is called by NUMERICAL.DATA, ALPHA.NUMERICAL.DATA, DEFINE.SUBSCRIPTS, and DATA.INPUT in order to pick up the data card images and scan them. NUMERICAL.DATA is used in DEFINE.SUBSCRIPTS to pick up numerical subscripts and in DATA.INPUT for numerical data. ALPHA.NUMERICAL.DATA is also used in DATA.INPUT for reading non-numeric or text-type data. Likewise, DEFINE.SUBSCRIPTS is used in DATA.INPUT for defining variable subscripts before data is stored into them. And finally DATA.INPUT itself is called in SINGLE.READ and ARRAY.READ for actually storing data into the proper variable names. Calls on SINGLE.READ and ARRAY.READ are assembled into the final ALGOL program at the proper places designated by the placement of input data in the HYDRO program.

The data read flow chart begins with the declarations of the variables used in the data read procedures. The function of each of these variables has been explained in Chapter 6. Since these are actually the first variables declared in the ALGOL program assembled from the HYDRO code, they must be considered reserved just like other variables used in HYDRO procedures. Therefore they should not be used when developing new procedures for insertion into the HYDRO library.

Following the declaration of variables, the NEXT.COLUMN procedure is declared. Its purpose is to read alphanumerically one by one the data card images at the end of the ALGOL program into a logic vector called CHAR. It then moves a pointer,COLM, across each card image, one column at a time, as the data are removed from the card.

Initially, in NEXT.COLUMN, the counter, COLM, is incremented by one across the CHAR vector. A check is then made if the end of the card image has been reached, that is, if COLM is greater the 80. If it is not, an exit is made from NEXT.COLUMN. If it is greater than 80, a new card image must be read into CHAR. This is done and the column pointer is reset to one. An exit is then made from the procedure.

Following NEXT.COLUMN the NUMERICAL.DATA procedure is declared as a REAL procedure. This means that the procedure itself will take on a REAL value, which will in most instances be stored into the REAL procedure DATA.INPUT.

NUMERICAL.DATA begins by decrementing the counter, COLM, since it will be incremented by NEXT.COLUMN at LLL1. NEGATIVE is set to FALSE, the value of the procedure is initially set to zero, and the variable, EXPO, is set to one. EXPO is a variable which represents the power of ten which must multiply the current value of NUMERICAL.DATA to give it its correct value.

NUMERICAL.DATA is always called when a number is expected in the columns to be scanned next on the data card image stored in CHAR. This number will either be an array element designator to be stored into one of the SUB variables, or an actual piece of numerical data to be stored into DATA.INPUT. At LLL1 the search is begun for the number. NEXT.COLUMN is first called to increment the column counter. A check is then made to see if this element of CHAR is blank. If it is, a return is made to LLL1 to move to the next column. If it is not blank, a check is made for a plus sign. If it is found, control goes back again to LLL1 to go to the next column. Then a check is made for a negative sign. If it is found, the variable NEGATIVE is set to TRUE, and control again returns to LLL1 to go to the next column. If a minus is not found, the procedure checks by a series of 10 successive statements to see if CHAR [COLM] contains one of the 10 digits. If one of these is found it is stored into the variable TEMP. In case a digit was not found, a check is made for a decimal point. When a decimal point is found, control passes to LLL2 to set the power of ten variable, EXPO, so that it will handle a fractional part. If no decimal point is found, a check is made for either a comma, a semicolon, or a right bracket. All three of these signal the end of the number, and if one is found, control passes to LLL3. If one of these is not found, then a digit was located above and a new value of NUMERICAL.DATA must be calculated. This is done by multiplying the old value by ten and adding the new digit found in TEMP.

Once the new value has been calculated, a check is made if the power of ten variable is less than one, that is, if the digits being processed are part of the fractional part of the number. If this is so, the power-of-ten variable, EXPO, is decremented by one. This means the final result will eventually be raised to one higher negative power of ten in order to push the decimal point back to the correct position. The procedure then goes back to LLL1 to pick up the next digit. If EXPO is not less than one, then the digits being processed are to the left of the decimal point. EXPO does not have to be changed, and control can immediately pass to LLL1 to look for the next digit.

The label LLL2 is reached when a decimal point is found in a number. In this case, EXPO is changed from 1 to 0 in order to prepare for the processing of the fractional part of the number. Control then returns to LLL1 to look for the digits to follow the decimal point.

The label LLL3 is reached when a comma, semicolon, or right bracket is found, signalling the end of the number. A check is made at this point to see if the power-of-ten, EXPO, is less than zero. If it is, a fractional part has been processed, and the number currently in NUMERICAL.DATA is too large by some power of ten. The decimal point is pushed to the left the correct number of places by multiplying NUMERICAL.DATA by 10 to the negative power EXPO. Then if NEGATIVE is TRUE, the sign of NUMERICAL.DATA is changed. After this, NUMERICAL.DATA contains the correct value of the number on the data card. Before an exit is made from the NUMERICAL.DATA procedure, the last 19 elements of the ALPH array are made blank. This must be done only in order to catch one of the data read errors given at label ERROR3. It has nothing to do with the storing of the proper number into NUMERICAL.DATA.

Following the declaration of NUMERICAL.DATA is the declaration for ALPHA.NUMERICAL.DATA. This procedure is called within the DATA.INPUT procedure, and its purpose is to store data alphanumerically into a vector called ALPH.

At the entrance to ALPHA.NUMERICAL.DATA the 20 words of the ALPH array are made blank. Then the ASTERISK variable is set to FALSE. It will remain in such a setting until the right asterisk at the end of the piece of logic data being read is found in the CHAR vector. Following the definition of ASTERISK, the counter,II, is set to one. This signifies that ALPH [1] is about to receive data, at the label LLL4. Each time a word of ALPH is completely filled with data and the right asterisk is not found in CHAR, II will be incremented by one, and control will return to LLL4 to read data into the next word of ALPH.

At LLL4, an iteration loop on JJ is entered; JJ begins at 1 and is incremented to 4. At each value of JJ, one of the four characters which will constitute the word of ALPH being processed is stored into that word. At each loop of the iteration, JJ is given a new value, and a check is made by the iteration statement to see if that value is greater than four.

If it is not then the present contents of ALPH are to be left-justified, and if the asterisk has not yet been found, another character is to be stored into ALPH.

First, the present contents of the ALPH word are shifted one character to the left by multiplying the logic word by 256. Then a check is made to see if the asterisk has been previously encountered in the CHAR array, that is, if ASTERISK is TRUE. If it is already TRUE, then the left-shifting of the contents of ALPH is all that is to be accomplished on the iteration, and JJ is incremented by means of the iteration statement. Once the ASTERISK is found, this cycle of left shifting and incrementing JJ is continued until JJ is beyond 4. This simply left-justifies the contents of ALPH if it consists of less than four characters.

If, on the other hand, the asterisk has not been found, a check is made to see if it is located in the present column of CHAR being scanned. If it is, the ASTERISK variable is made TRUE, and the COLM counter is given the value of the column where the asterisk was found. A return is then made to the iteration statement where the present contents of ALPH will be left-justified if JJ has not yet gone beyond 4.

If the column of CHAR being scanned does not contain an asterisk, then the contents of that column are to be stored into the rightmost bit of the ALPH word. This is done by logically adding the contents of that column to the present contents of ALPH [II] and storing the result into ALPH [II] . Control then returns to the iteration statement to examine the next character of CHAR, or possibly to store into the next word of ALPH, if JJ has gone beyond 4.

When the iteration statement is completed, that is, JJ does go beyond 4, a check is made to see if the right asterisk of the piece of logic data being read has been encountered yet in CHAR. If it has not, the COLM counter is incremented by 4 since an entire word has been stored into ALPH, and the counter on ALPH, II, is incremented by one since the next word of ALPH is about to be filled. Control then returns to LLL4 to fill that word. If the asterisk was found during the processing of the previous ALPH word, then the entire piece of logic data in CHAR has been stored into the words of the ALPH array, and the principal task of ALPHA.NUMERICAL.DATA has been completed.

At LLL5 code is executed to move the pointer on the CHAR array from the position of the right asterisk to the position of the comma or semicolon following the piece of alphanumeric data just read. The procedure NEXT.COLUMN is called at LLL5, and if a comma or semicolon is not found in the new column of the CHAR vector, NEXT.COLUMN is called again. This loop continues until a comma or semicolon is found. An exit is then made from the ALPHA.NUMERICAL.DATA procedure.

After the declaration of ALPHA.NUMERICAL.DATA, the procedure DEFINE.SUBSCRIPTS is declared. The purpose of this procedure is to redefine the three subscript variables, SUB1, SUB2, and SUB3, each time a new data variable is about to be read.

DEFINE.SUBSCRIPTS begins by setting the DEFINED switch to FALSE. If it remains FALSE until the end of DEFINE.SUBSCRIPTS, then SUB1, SUB2, and SUB3 will all be set to one. On the other hand, if DEFINED is made TRUE at some point in DEFINE.SUBSCRIPTS, it indicates that a subscripted variable was read and the subscript variables were DEFINED by calls on the NUMERICAL.DATA procedure.

After DEFINED is made FALSE at the start of DEFINE.SUBSCRIPTS, a call is made at LLL6 on NEXT.COLUMN in order to move the column pointer. A search is then begun on the data card image in CHAR for a left arrow or a left bracket following the name of a data variable. The left bracket will be found whenever the input data variable is subscripted, i.e. a particular numerical array location is designated following the variable name on the data card. As long as one of these two is not found, the program cycles back to LLL6 to look for one in the next column. When finally a left arrow or left bracket is found, a check is made to see if it is the left bracket. If the left bracket is found instead of the left arrow, then code must be executed which will read the numerical array locations into SUB1, SUB2, and SUB3 by means of calls on the procedure NUMERICAL.DATA. If the left arrow is found instead, this code is skipped.

The code for reading array locations begins by calling NEXT.COLUMN. NUMERICAL.DATA is then called and the result is stored into SUB1. A check is then made by looking for a comma to see if a second subscript number is given. If it is, NEXT.COLUMN is called and SUB2 gets the results of a second call on NUMERICAL.DATA. A check is then made for a third number. If its presence is indicated by a comma, NEXT.COLUMN is called again and SUB3 gets the third call on NUMERICAL.DATA. If at any time in this sequence a comma is not found, or if finally all three subscript variables are defined, then DEFINED is made TRUE and control passes back to LLL6 to locate the left arrow following the sbuscript brackets.

When a left arrow is finally found at LLL6, the code executed when a left bracket is found is skipped, and a check is made on DEFINED to see if it is TRUE. If it is FALSE, all three subscript variables are set to one. If it is TRUE, the subscript variables are already defined, and control passes out of the DEFINE.SUBSCRIPTS procedure.

The DATA.INPUT procedure is declared after the DEFINE.SUBSCRIPTS procedure.

7.2.8

This is the principal procedure for reading data, and it calls all of the others declared before it. It is called itself by SINGLE.READ and ARRAY.READ.

Initially, DATA.INPUT sets FINISHED to FALSE. The value of FINISHED is a signal used in the reading of arrays. As long as FINISHED is FALSE, data reading continues for the successive elements of a particular array. When a semicolon is finally found on a data card FINISHED is made TRUE and no more data are read into that array.

At the label LLL7, code is located which actually gives data values to the DATA.INPUT procedure, or, in the case of alphanumeric data, to the vector ALPH. This is accomplished by calls on NUMERICAL.DATA or ALPHA.NUMERICAL.DATA. First, NEXT.COLUMN is called to move the column pointer beyond the position of the left arrow or comma. Then a check is made to see if a blank column has been found. If it has, control passes back to LLL7 to move over another column. When finally something is encountered besides a blank, a check is made to see if it is an asterisk. If an asterisk is found, a call is made on ALPHA.NUMERICAL.DATA to read that type of data into the ALPH vector. If an asterisk is not found, then regular numerical data is located on the card. A call is made on NUMERICAL.DATA, and its value is given to DATA.INPUT.

Once the piece of data has been processed, be it numeric or alphanumeric, a check is made for a semicolon which signals the end of data for a particular variable. If it is not found, then more data follows for the same variable, and control passes out of the DATA.INPUT procedure. If, on the other hand, a semicolon is found, then certain tasks must be performed in preparation for the reading of the next variable.

First, FINISHED is made TRUE to show that all the data has been read for the present variable. Then at LLL8 a call is made on NEXT.COLUMN to increment the column counter in search of the next variable name. Each time a blank is found in the new column, control returns to LL18 and NEXT.COLUMN is called again. Finally, the next variable name is reached. When this occurs, the column counter is decremented and the first four letters of the variable name are read into ABCD by means of NAME and READ statements. This enables a check to be made to see if the correct amount of data has been given to each succeeding variable.

Following the reading into ABCD, checks are carried out to see if the procedure DEFINE.SUBSCRIPTS should be called. Each time a single variable has been read into, be it numeric or alphanumeric, DEFINE.SUBSCRIPTS should be called. It can also be called if a numeric array has been read into. Only in the case where an alphanumeric array has been receiving data should the call on DEFINE.SUBSCRIPTS be skipped. In that case, the DEFINE.SUBSCRIPTS call is located in the ARRAY.READ procedure. These checks for DEFINE.SUBSCRIPTS are made by examining ALPH [1] and SUB1. If ALPH [1] contains dollars, then numerical data has been read and DEFINE.SUBSCRIPTS can be called. Also, if SUB1 is still equal to one, then a single variable has been defined, and once again, DEFINE.SUBSCRIPTS can be called. After the new subscript variables are defined by DEFINE.SUBSCRIPTS, control passes out of DATA.INPUT. If it is found that the DEFINE.SBUSCRIPTS procedure should not be called, then control passes out of DATA.INPUT at that point.

The SINGLE.READ and ARRAY.READ procedures are declared following DATA.INPUT. Both of these procedures are parametered.

The four parameters of SINGLE.READ are VAR1, VAR2, VAR3 and TYPE. The first three are dummy names for general REAL, INTEGER, and ALPHANUMERIC (or LOGIC) single variables. When the procedure is called to store data into a single variable, the name of the variable is always substituted for all three of these dummy parameters. The fourth parameter,TYPE, indicates which type of variable (real, interger, or alphanumeric) is really being read into when SINGLE.READ is called. It is given the values 1,2, or 3, respectively to indicate these types.

SINGLE.READ begins by declaring its parameters, and then it checks the value of TYPE. If TYPE is equal to 1 it knows that the variable to be read is a REAL variable. It therefore calls DATA.INPUT and stores its numerical value into VAR1, the REAL variable. If TYPE is not equal to one, the procedure checks if it is equal to 2. If so, DATA.INPUT is called, and the result stored into VAR2. If TYPE is not equal to 1 or 2, then the single variable is alphanumeric, and TYPE will equal 3. When this is the case, DATA.INPUT is called, storing one alphanumeric data word into ALPH [1] . This word is then given to VAR3. The ALPH [1] variable is then reset to $$$$ since this is the state it must have following the reading of any variable. An exit is then made from SINGLE.READ.

ARRAY.READ works on the same principles as SINGLE.READ. It is more complicated, however, because one, two, or three dimensions may be involved for each type of array, and because more than one piece of data must generally be read by the procedure.

The parameters of ARRAY.READ are VAR1, VAR2, VAR3, TYPE, FIRST4, BND1, BND2, and BND3. The first four of these parameters serve the same function as they did for SINGLE.READ. The others are used as follows. FIRST4 always contains the first four letters of the variable name. A comparison of the contents of FIRST4 with ABCD indicates whether an error has been made in inputting data to certain arrays. The three parameters, BND1, BND2, and BND3, always contain the three bounds of the array being read. When an array is less than three dimensional, the appropriate bounds are set to zero.

ARRAY.READ begins by declaring all of its parameters, and then it checks FIRST4. If FIRST4 is not equal to ABCD an error message is given from ERROR1. At LLL11 the procedure then checks if the array to be read is of type REAL, i.e., if TYPE is equal to 1. If this is not so it skips the code for the reading of REAL arrays, and checks for an INTEGER array. If, however, a real array is to be read, the procedure checks if BND2 = 0. If so, the array is one dimensional, and data is stored into the SUB1 element of the one dimensional REAL array variable, VAR1, by a call on DATA.INPUT. Control then passes to LLL12 to alter SUB1.

If on the other hand the array is not one dimensional, a check is made to see if it is two dimensional, i.e. if BND3 = 0. If so, the REAL, two-dimensional array variable is defined by a call on DATA.INPUT and control passes to LLL12 to alter SUB1 and SUB2.

Finally the array may be three dimensional. In this case, the three dimensional REAL array variable is defined by DATA.INPUT, and control again passes to LLL12 to alter the subscripts.

The code executed when TYPE is equal to 2, i.e. when an INTEGER array is read into, is extremely similar to that executed for a REAL array. The only difference is that in all cases of dimension, the INTEGER array variable, VAR2, is defined instead of VAR1.

If TYPE is equal to 3 an ALPHANUMERIC array must be defined. In this case, two additional variables are declared: I, a dummy variable, and BOUND, the number of 4 letter computer words contained in each of the HYDRO words of alphanumeric data to be read. With these variables declared the DATA.INPUT procedure is called in order to store the 4 letter alphanumeric words contained in the one piece of HYDRO alphanumeric data into the ALPH array.

The procedure then checks the bounds of the array in order to define the BOUND variable prior to storing the words in ALPH into the alphanumeric variable, VAR3. If the array is one dimensional, BOUND is set to one. If it is two or three dimensional, it is set to BND2 or BND3, both of which are integers.

Following the definition of BOUND, an iteration is carried out in which the BOUND number of words in ALPH are stored into VAR3. If the array is one dimensional, the one word in ALPH [1] is stored into VAR3 [SUB1] . If it is two or three dimensional the BOUND words in ALPH [I] are stored into VAR3 [SUB1,I] or VAR3 [SUB1, SUB2, I] . When the iteration is complete, all the words of ALPH should have been given to VAR3. A check is made on ALPH to see if this has been done. If any extra words have been given to ALPH, i.e. if the one piece of HYDRO alphanumeric data just read contained more 4 letter computer words than were designated by the final numeric bound of the array, then an error message is printed from ERROR4. If this error is not encountered, control passes to LLL12 to alter the subscripts, SUB1, SUB2, and SUB3.

At LLL12 a check is first made to see if all data have been read for the particular variable being processed. If reading is FINISHED, control passes to LLL14. If reading is not complete, the subscript variables must be altered prior to reading of the next piece of data.

A check is first made to see if SUB1 is the only subscript to be altered. This would be so far a one dimensional REAL or INTEGER array or a two dimensional ALPHANUMERIC array. If this is the case, SUB1 is incremented and control passes to LLL13. If this is not so, a check is made to see if SUB2 must be altered. This would be so far a two dimensional REAL or INTEGER array or a three dimensional

ALPHANUMERIC array. If this case holds, SUB2 is incremented, and a check is made to see if it has exceeded its bound, BND2. If so, it is reset to 1 and the first sbuscript, SUB1 is incremented. That is, in effect, moving to the next row of the array. Control again passes to LLL13. If SUB2 is not to be changed, then finally a check is made to see if SUB3 is to be incremented. This would be so if data for a three dimensional REAL or INTEGER array was being read. If this situation prevails, SUB3 is incremented and a check is made to see if it has exceeded its beound, BND3. If it has not, control passes to LLL13. If it has, it is reset to one, and SUB2 is incremented. Now it too must be checked to see if it has exceeded its bound, BND2. If not, control goes to LLL13. If it has, it likewise is reset to 1 and SUB1 is incremented by one. Control in this case also goes to LLL13.

At LLL13 a final check is made to see if the first array subscript, SUB1, has exceeded its bound, BND1. When this happens, too many pieces of data have been given to the array and an error message is printed at ERROR2. If this is not so, control returns to LLL11 to read the next piece of data.

At LLL14 final checks are made to see if the correct type of data has been given to the data variable just read. If TYPE is equal to 1 or 2 for a REAL or INTEGER array, then no data should have been read into the ALPH array by a call on ALPHA.NUMERICAL.DATA. Therefore ALPH [1] should still be equal to $$$$. If it is not, an error message is printed at ERROR3. If it is, control passes out of ARRAY.READ at LLL15. On the other hand, if TYPE is equal to 3 and an ALPHANUMERIC array was read, ALPH [1] should not equal $$$$. Therefore, if it does, an error is again given at ERROR3. If it does not, it is set to $$$$ in preparation for the reading of data into the next variable. DEFINE.SUBSCRIPTS is also called at this point to prepare

SUB1, SUB2, and SUB3 for the reading of a new variable. Control then passes out of the ARRAY.READ procedure.

Following the declaration of the seven data read procedures, there is a short section of code which is placed along with them at the head of every ALGOL program assembled from HYDRO. Because this is the first code in the ALGOL program which is not some sort of a declaration, it is the first code which is executed. Its purpose is to DEFINE.SUBSCRIPTS, and to store into ABCD and ALPH [1] for the first time, before data read code is encountered later in the ALGOL program.

First, the pointer on the CHAR vector, COLM, is set to 80 so that the first data card image will be read into CHAR by NEXT.COLUMN at LLL9. Then a check is made for a blank at the beginning of the card. If one is found, control returns to LLL9 to move to the NEXT.COLUMN. This loop continues until something is found on the data card. When this happens, the column counter is decremented by one, and by means of NAME and READ statements, the first four letters of the name found on the data card are read into ABCD. Then a check is made to see if the word "END;" has been stored into ABCD rather than the beginning of a data variable name. This will usually happen because six cards of ENDs are placed behind the ALGOL program before the data is placed there. This is usually more than is required. In such a case, the ALGOL program considers several of these cards of ENDs to be data cards. When this happens, the COLM counter is set to 80 and control returns to LLL9 where the next card is read into CHAR, and the first four letters on it are read into ABCD. This process continues until all cards of ENDs are bypassed, and an actual variable name is located on the first data card.

Once ABCD has been properly defined, the procedure DEFINE.SUBSCRIPTS is called to read into the SUB1, SUB2, and SUB3 variables for the first time.

Following this, the first element of the ALPH array is set to dollar signs. This word of ALPH is used as a signal in all of the data reading to follow; it permits the detection of several errors. The only way ALPH [1] can be later changed to anything besides dollar signs is for a call to be made on the procedure ALPHA.NUMERICAL.DATA. Therefore, the condition of ALPH [1] can be used as a signal to tell whether or not this procedure has been called. This tells if alphanumerical data is being given to alphanumerical data variables, and if numerical data is being given to numerical data variables. Once the ALPH [1] variable has been defined for the first time, control is transferred across the data read error nessages by the command GO TO LLL10. At this label the remainder of the ALGOL program assembled from the HYDRO input code is run.

8. AUGMENTATION OF HYDRO

The following Chapter describes the method for augmenting the Carnegie Tech HYDRO system by the addition of a new procedure. The principles underlying this method also apply when the existing system must be corrected or altered, or when more than one procedure is being added.

This Chapter is by necessity oriented toward the Carnegie Tech system. All the unique aspects, however, of this version of HYDRO are explained in Appendix I. Furthermore, the principles underlying the alteration of this system would apply for any HYDRO system, although the mechanics might be slightly different.

Initially, this Chapter explains the simple regulations governing the different types of variables used in the procedures incorporated into the HYDRO library. These regulations, which are the result of the specifications established for HYDRO as outlined in the Users Manual (8), must be understood by anyone designing procedures for incorporation into the HYDRO system. Secondly, a step-by-step method of designing procedures so that they can be easily inserted into the system is established. Finally, the process by which a procedure enters the library and the code for processing it is put into the compiler,is outlined.

8.1 Rules Governing HYDRO Variables

Procedures in the HYDRO language employ two types of variables:

1. Single Variables
2. Array Variables

Each of these two categories has the same three sub-types:

1. Global Read-In
2. Global Calculated
3. Local

The regulations governing the use of these six types of variables in HYDRO procedures are outlined below. In these specifications, the word "declaration" refers to the place in a program where storage is set aside for a particular variable, i.e. the point where the computer is informed that a certain variable requiring a specified amount of computer space will be used. The word "definition" refers to the point where a variable is first assigned a numeric or alphanumeric value.

Global Read-In Single Variables

- All single variables which are read in to any procedure are automatically global variables. HYDRO has no local read-in variables.

- Declaration: Read-In single variables are declared automatically by the compiler. They are all declared, whether they will be used or not, in one of the outer blocks of every program assembled and run by HYDRO. Therefore, none of these variables are declared within the individual procedures which use them. This slightly wasteful process is employed in order to simplify the compilation of single variables which are read in.

- Definition: Read-in single variables are defined by a call on SINGLE.READ assembled at the proper place by the compiler into the ALGOL program. These variables may also be redefined within a procedure by means of an assignment statement.

Global Calculated Single Variables

- These single variables are never read in, but are global to more than one procedure; i.e. their values have to be preserved for use in another procedure.

- Declaration: Like the global read-in single variables, these are all declared automatically by the compiler in the outermost block of every program assembled and run by HYDRO. None are declared within individual procedures.

- Definition: Global calculated single variables are defined by an assignment statement within one or several of the procedures.

Local Single Variables

- Local single variables are used in only one particular procedure. It makes no difference if their values are lost (i.e. they become undefined) once that procedure has been completely executed.

- A variable which is local can never be read into in HYDRO.

- Declaration: Local single variables are declared at the beginning of the procedure in which they are used.

- Definition: Local single variables are defined by one or several assignment statements within the procedure to which they are local.

Global Read-In Arrays

- There is no such thing in HYDRO as a local read-in array. If an array is to be read into, it must automatically be global.

- Bounds of read-in arrays are always global single variables or integers. The global single variables are declared automatically at the beginning of each program. They are defined in one of three ways. They ordinarily get their values by means of a data read-in previous to the array declaration. They may also be defined by an assignment statement in a procedure preceding the declaration of the array, or in some cases by an assignment statement inserted into the assembled program by the compiler prior to the insertion of the array declaration.

- If an array bound has to be defined by a data-read, it must always be given as data by the user before its array is given as data.

- Declaration: The declarations of read-in arrays are assembled automatically by the compiler into the ALGOL program. When an array name is encountered on a data input card by the HYDRO compiler it checks to see if that array has ever been declared before in the program being assembled. If it has not, a declaration for the array is assembled into the program, and this declaration is followed by the call on ARRAY.READ which will define the array. If the array has been declared previously, only the call on ARRAY.READ is inserted.

- Definition: As stated above, the ARRAY.READ call inserted by the compiler will cause the array to be defined. This read-in does not actually take place, however, until the running of the total assembled program.

Global Calculated Arrays

- If an array requires reading for any one of the HYDRO procedures, it is of class "read-in" for all the procedures, even though in some it is only calculated. Thus, the class of global calculated arrays are those which are never read into. They are global, however, because their values are required in more than one procedure.

- Bounds of global calculated arrays are always global single variables. As in the case of global read-in arrays, these are declared automatically by the compiler, and are defined in one of three ways: by a data read in, by an assignment statement in a previous procedure, or by an assignment statement inserted by the compiler.

- If a bound is to be read in, no error will occur as long as the user enters the bound as data for the first procedure which uses the global array, or prior to this. This insures that the bound is defined before the declaration of the global array.

- Declaration: Global calculated arrays, like read-in arrays, have their declaration assembled automatically by the compiler. This is accomplished as follows: For each procedure called by the user, the compiler contains a special section of code which assures that all variables needed for that particular procedure are declared and defined. If it is found that certain global calculated arrays are not declared which are needed, the compiler assembles these array declarations before it assembles the code for the procedure.

- Definition: Global calculated arrays are defined by assignment statements within one or more of the procedures to which they are global.

Local Arrays

- Local arrays are never read into, and never used globally. They are used in only one particular procedure.

- Declaration: Local arrays are declared within the procedure to which they are local. Thus, the declarations of local single variables and local arrays are the only declarations which will be found inside the procedure bodies themselves. All other declarations are assembled global to procedures by the compiler.

- Definition: Local arrays are defined by assignment statements within the procedure.

- The values stored into local arrays may be typed out or used within the procedure, but are never carried over for use in another procedure.

- Bounds to local arrays may be either local or global. If the bound is local it must be both declared and defined in the procedure at a position above the declaration of the local array. This should be assured by the programmer in designing the procedure. If the bound is global, the compiler will declare it and check if it has been defined before the procedure is assembled.

8.2 Design of a Procedure for Insertion into HYDRO

General Principles

Several general principles should be adhered to in the design of all procedures to be incorporated into the HYDRO system. These are established in order to foster uniformity as well as flexibility in the procedures of the language, and to simplify the task of inserting them into the library. These principles are listed below.

1. Large tasks should be broken into a set of smaller tasks. In other words, large procedures should be dissected, if possible, into smaller procedures, as long as the smaller tasks have merit as procedures in their own right. The test of the merit of a small procedure is whether or not it can be called by itself to carry out a particular task, or whether it can be used as part of the code of another larger procedure besides the one being programmed.

An example of this concept can be found in the design of procedures to carry out frequency analyses of streamflow data. Several large procedures could be designed which would satisfy the needs of many users in this area. These procedures, however, would be lengthy and rigid. Far more users could be satisfied in a much more general way be the design of a greater number of small procedures, each executing one of the simple tasks which can be combined to perform a detailed frequency analysis. Programming in this manner builds flexibility into the system.

2. Procedures to be called by the user should always be parameterless, so that they can be employed by simply calling the procedure

name. A case will sometimes arise, however, where it appears that a general process should be programmed as a procedure with parameters which can be called by many other procedures. When this occurs, other procedures with different names should be designed without parameters for use by the user. These procedures can then contain calls on the parametered procedure with an appropriate set of arguments. An example of this is the STANDARD.DEVIATION procedure. It is desirable to have a procedure to compute a standard deviation which can be used throughout the HYDRO system in any of its procedures. This requires a procedure with dummy parameters which can be called in any other procedure by supplying a set of real parameters. However, in the set of programs for FREQUENCY ANALYSIS it is desirable to have a procedure which the user can call to obtain the standard deviation of the set of events he wishes analyzed. This requires a procedure with no parameters. The solution, as suggested above, is to design two procedures, one with parameters and one without parameters. The second procedure calls the initial one. Thus, the first procedure is declared:

```
  PROCEDURE  STANDARD.DEVIATION (VECTOR, LENGTH, MEAN, STANDEV)  ;
```

and has general code to compute the standard deviation of any vector of numbers with a length and a mean value of the numbers. This procedure can be used anywhere that a standard deviation is required. The second procedure is declared

```
            PROCEDURE  STANDARD.DEVIATION.OF.EVENTS ;
```

and has a call within it with real parameters as follows:

```
   STANDARD.DEVIATION  (SERIES, TOTAL2, MEANVALUE, DEVIATION)  ;
```

This procedure is only used by the HYDRO user when it is desired as part of a frequency analysis.

3. The variables used in HYDRO procedures, especially the variables with which the user comes into contact, should be descriptive of what they actually represent. Variable names should be at least 4 characters long, but should not exceed 15 characters in length. Six to eight characters is the usual length.

4. The same variable should never be used to represent two different things in the same program, or even in different programs. Therefore, when one designs a procedure, he must be sure he is not using variable names which have been used previously as read-in or global variables in any other procedure. This is especially important with regard to variables which will be used for read-in data. An identical principle holds for procedure names. No two procedures should have the same name. On the other hand, when a variable name does represent the same thing in two different procedures, it should be given the same variable name. This is especially true for global variables whose values are to be transferred from procedure to procedure.

5. Some variables are not of sufficient importance to warrant having a descriptive variable name. Integer counters fall into this category. Single letter variables should be used for these, especially in iteration statements. In general, the letters I, J, K, L, M, N, P, Q, R, S, and T are used. Some real variables are also of small importance. The dummy variables D1, D2, D3, D4, D5, D6, D7, D8, and D9, are used for these. Both these sets of variables, the integer counters and the dummy variables, are declared globally by the compiler, so that they do not have to be continually declared within the procedures. No dummy arrays are declared globally. These should be declared within the procedure in which they are used.

6. The declaration of variables for a procedure should follow a specific order, just for the sake of clarity. In any block of declaration statements the OWN variables should be declared before the REGULAR variables. SINGLE variables should also be declared before ARRAYS, and BOOLEAN, LOGIC, and INTEGER variables should be declared before REAL variables. Thus, the order in declaration is as follows:

OWN BOOLEAN
OWN LOGIC
OWN INTEGER
OWN REAL
OWN BOOLEAN ARRAY
OWN LOGIC ARRAY
OWN INTEGER ARRAY
OWN REAL ARRAY
BOOLEAN
LOGIC
INTEGER
REAL
BOOLEAN ARRAY
LOGIC ARRAY
INTEGER ARRAY
REAL ARRAY

7. Labels, as a rule, do not have to be descriptive, because the user never comes into contact with them. Unless it is desired to give a label a descriptive title, the letter L followed by a number should be used for a label. L1, L2, L3, L25, L62, and L158 are all acceptable labels. Usually the programmer begins his labels at L1 and continues incrementing the number following the L as long as labels are needed. A label need never be repeated if the programmer does not wish it to be, since the numbers are limitless.

8. Every procedure has a hierarchy of commands, and this hierarchy should be indicated by the 3-space indentation of code. This indentation usually comes after the word BEGIN, or after a FOR statement, or when the line to be indented is of less importance or is under control of the outer line. Indentation may also be used anywhere that it will enhance

clarity. The following is an example of indented code:

```
BEGIN
IF ISOHYET3[I] = 0 THEN BEGIN
   P ← P + 1;   Q ← Q + 1;
   IF ZONE3[M] = ZONE3[K]   THEN BEGIN
      IF PLANIMETER4[I] = 0   THEN GO TO L91;
      FOR R ← 1 STEP 1 UNTIL TOTAL12 DO
         MASS4[I,R] ← SUMOFP3[Q,R] *
         STATION4[I];
      NOS4[I] ← 0;
      GO TO L96;
      L91:  IF SUMSIGMA[I] ≠ 0   THEN T ← T-1;
   END;
END;
etc.
```

9. As a general rule, only one command should be placed on a card, unless the commands are very short. Of course, longer commands will require more than one card, and can be continued on succeeding cards. Spaces should be skipped within commands only where it is necessary for clarity and the correct running of the program.

Step By Step Design

Adhering to the foregoing principles, the following step-by-step process should be followed in designing a procedure which is to be inserted into the HYDRO language.

1. DECLARE INTEGER COUNTERS AND DUMMY VARIABLES. That is, declare I, J, K, L, M, N, P, Q, R, S, T, and D1, D2, D3, D4, D5, D6, D7, D8, and D9. Any of these can then be used in the procedure being developed, and because they are declared automatically by the compiler, they can be forgotten when the procedure being developed is inserted into the system.

2. DECLARE GLOBAL BOOLEAN, LOGIC, INTEGER, AND REAL SINGLE VARIABLES. Most variables, both single and array variables, will be global to the procedure. Any input variable, or any variable whose value is transferred from another procedure, or any variable whose value must be kept for use in a later procedure, will be a global variable. These should all be declared global to the procedure body. The only variables which should be declared inside the procedure body are those which are local to that procedure only.

The statements used to declare the global single variables are of the form:

```
BOOLEAN  Variable Name List;
LOGIC  Variable Name List;
INTEGER  Variable Name List;
REAL  Variable Name List;
```

Labels which are used as the parameters of procedure calls should also be declared at this time.

3. READ DATA INTO SOME OF THE GLOBAL SINGLE VARIABLES. At this point, data should be read, where necessary into the global single variables. In many cases these will be the bounds of the arrays which are about to be declared. No data reads should be done inside the procedure body,

since all data reads will be handled automatically by the compiler when the procedure is incorporated into the HYDRO system. In the ALGOL program being prepared for HYDRO, these data reads are done by the usual READ statements of ALGOL.

4. DECLARE GLOBAL BOOLEAN, LOGIC, INTEGER, AND REAL ARRAYS. At this point, most global arrays can be declared since their bounds were established by Steps 2 and 3. These are declared by statements of the form:

```
BOOLEAN ARRAY  Variable Name List;
LOGIC ARRAY  Variable Name List;
INTEGER ARRAY  Variable Name List;
REAL ARRAY  Variable Name List;
```

5. READ DATA INTO SOME OF THE GLOBAL ARRAYS. After the declaration of the arrays, data should be read into those which are to receive data inputs. This is accomplished by ALGOL READ statements, or by the use of READ statements enclosed within FOR statements.

6. USE THESE GLOBAL ARRAYS TO CALCULATE THE BOUNDS ON OTHER GLOBAL ARRAYS. It sometimes happens that the bounds of some global arrays cannot be read in but must be calculated from the existing values of other global arrays. In this case, the former arrays cannot be declared until the latter arrays are declared and read into, and the bounds calculated from the data which has been read. An example of this situation is found in the DEPTH-AREA-DURATION procedures in HYDRO. In one of these data is first read into three arrays, ISOHYET3, PLANIMETER3, and ZONE3. Using the values in these arrays, the bounds REPEAT1 and REPEAT2 are calculated. These bounds are then used in the declaration of several other global arrays used by the procedure.

7. DECLARE THE OTHER GLOBAL ARRAYS WITH THE BOUNDS JUST CALCULATED. Once the bounds are calculated in Step 6, the arrays depending upon those bounds can be declared in the usual manner.

8. DECLARE ANY PROCEDURES CALLED BY THE PRINCIPAL PROCEDURE. In some cases the procedure being designed makes calls on other procedures. These others must therefore be declared prior to the declaration of the principal procedure. If these other procedures happen to have any material which must be global to them, it must be declared prior to their declaration according to the rules laid down in Stpes 2 through 7. So the order of declarations should be:

GLOBAL MATERIAL TO PRINCIPAL PROCEDURE
GLOBAL MATERIAL TO OTHER PROCEDURES
BODIES OF OTHER PROCEDURES
BODY OF PRINCIPAL PROCEDURE

9. DECLARE THE BODY OF THE PRINCIPAL PROCEDURE. After all procedures are declared which are used by the principal procedure, it can be declared itself. The procedure body should be written in such a way that it makes use of the global material declared and read prior to it in order to compute results. In general all the printing of computed results takes place within the body of the procedure.

10. CALL THE PRINCIPAL PROCEDURE. After the declaration of the procedure, the final task is to call it.

Example

The above ten steps should be followed in designing a procedure to be incorporated into the HYDRO system. The following example illustrates their use in the design of a program to execute an imaginary procedure, PROC.N. COMMENT statements throughout the program illustrate where the ten steps in the design were employed. Dummy names are used in the example to represent logic, integer, and real single variables and arrays. These names are of the form LOG1, INT1, REL1, LOGARY1, INTARY1, and RELARY1. In an actual procedure, variable names with a meaning applicable to the procedure would be used in place of these. The example program is as follows:

```
BEGIN
COMMENT    DESIGN OF IMAGINARY PROCEDURE - PROC.N ;
COMMENT NO.1 - DECLARE COUNTERS AND DUMMYS ;
   INTEGER   I, J, K, L, M, N, P, Q, R, S, T ;
   REAL   D1, D2, D3, D4, D5, D6, D7, D8, D9 ;
COMMENT NO.2 - DECLARE GLOBAL BOOLEAN, LOGIC, INTEGER, AND
   REAL SINGLE VARIABLES ;
   LOGIC   LOG1, LOG2, LOG3 ;
   INTEGER   INT1, INT2, INT3, INT4, INT5, INT6, INT7 ;
   REAL   REL1, REL2, REL3, REL4, REL5, REL6 ;
COMMENT NO.3 - READ DATA INTO SOME OF THE
   GLOBAL SINGLE VARIABLES ;
   NAME   (LOG1, LOG2, LOG3) ;
   READ   (<W, 4A, 1B, 4A, 1B, 4A >) ;
   NAME   (INT1, INT2, INT3) ;
   READ   (<W, 1D, 1B, 3D, 1B, 2D >) ;
   NAME   (REL1, REL2, REL3, REL4) ;
   READ   (<W, 6D, 1B, 4D, 1B, 5D, 1B, 6D >) ;
COMMENT NO.4 - DECLARE GLOBAL BOOLEAN, LOGIC, INTEGER,
   AND REAL ARRAYS ;
   BEGIN
   LOGIC ARRAY  LOGARY1 [1:INT1,1:3]  , LOGARY2 [1:INT2 ] ;
   INTEGER ARRAY  INTARY1 [1:INT3] ,  INTARY2 [1:INT2 ]  ;
   REAL ARRAY  RELARY1 [ 1:INT1,1:INT2 ] ,
      RELARY2[1:INT2 ] , RELARY3 [1:INT3]  ;
```

```
8.2.10      COMMENT NO.5 - READ DATA INTO SOME OF THE
              GLOBAL ARRAYS ;
              FOR I ← 1 STEP 1 UNTIL INT1 DO BEGIN
                 NAME    (LOGARY1 [I,1] , LOGARY1 [I,2] , LOGARY1 [I,3] ,
                    J ← INT2(RELARY1 [I,J])) ;
                 READ    ( < W, 4A, 4A, 4A > , ← INT2 < 2B, 4D > ) ;   END ;
              FOR I ← 1 STEP 1 UNTIL INT2 DO BEGIN
                 NAME    (LOGARY2 [I] , RELARY2 [I] ) ;
                 READ    ( < W, 4A, 2B, 6D > ) ;   END ;
            COMMENT NO.6 - USE THESE GLOBAL VARIABLES
              TO CALCULATE BOUNDS ON OTHER GLOBAL ARRAYS ;
              INT4 ← INT2/6 ;
              INT5 ← + (INT1/INT4) + 1 ;
              INT6 ← 0 ;
              FOR I ← 1 STEP 1 UNTIL INT2 DO
                 IF RELARY2 [I]  >  REL2 THEN
                 INT6 ← INT6 + 1 ;
            COMMENT NO.7 - DECLARE OTHER GLOBAL
              ARRAYS WITH THE BOUNDS JUST CALCULATED ;
              BEGIN
              INTEGER ARRAY   INTARY3 [1:INT6,1:INT4]  ;
              REAL ARRAY    RELARY4 [1:INT5,1:INT2] ;
            COMMENT NO.8 - DECLARE ANY PROCEDURES
              ALONG WITH MATERIAL GLOBAL TO THEM WHICH
              ARE CALLED BY THE PRINCIPAL PROCEDURE ;
              BEGIN
              LOGIC   LOG4,LOG5 ;
              INTEGER   INT8,INT9 ;
              REAL  REL7,REL8,REL9 ;
              NAME  (INT8,INT9) ;
              READ  ( < W,2D,1B,2D > ) ;
              BEGIN
              LOGIC ARRAY   LOGARY3 [1:INT8] ;
              INTEGER ARRAY   INTARY4 [1:INT9] ;
              REAL ARRAY   RELARY5 [1:INT8] ;
              PROCEDURE   PROC.A ;
                 BEGIN

                (BODY OF PROCEDURE PROC.A) ;

                 END ;
            COMMENT NO.9 - DECLARE THE PRINCIPAL PROCEDURE ;
              PROCEDURE   PROC.N;
                 BEGIN

                (BODY OF PROCEDURE PROC.N) ;

                 END ;
            COMMENT NO.10 - CALL THE PRINCIPAL PROCEDURE ;
              PROC.N ;
            END ; END ; END ; END ; END ;
```

8.3 Insertion of a New Procedure into HYDRO

Preparation

Once a procedure has been designed according to the rules established previously it is not a difficult task to insert it into the HYDRO system. The description of this process, however, involves many of the terms peculiar to the Carnegie Tech HYDRO system. These are explained for the uninitiated reader in Appendix I. Facilities which might duplicate those at other computer centers are described and discussed in Chapter Nine.

The sample procedure, PROC.N, will be used as an example in the following description. All of the steps necessary to insert it into the HYDRO system will be illustrated.

First, additions must be made to the language, i.e. AND RECORD TWO, in the Carnegie Tech system, and then to Phases One and Two of the compiler in storage, which is AND RECORD ONE. After the compiler Phases of RECORD ONE have been changed, they have to be stored onto the tape segments which comprise the compiler in use.

The additions to the language library consist of the following: first, all new global single variables must be declared. These are followed by calls on SINGLE.READ for the new read-in single variables. Next, new read-in arrays are declared, with each declaration followed by an appropriate call on ARRAY.READ. Then the code for declaring global arrays which are not of the read-in type is stored into the section of the library containing this type of code. And finally, the body of the new procedure is stored into the library.

These insertions into RECORD TWO are made by submitting an AND program of ALTER and INSERT AFTER statements.

Once AND RECORD TWO has been changed, additions can be made to the various segments of the compiler in storage on AND RECORD ONE by means of another AND program. The first to be changed is Phase One. The cards containing calls on PR, SR, and A^R for recognizing the new procedure and and data variable names must be inserted into the old segments of Phase One on AND RECORD ONE. Then Phase Two is changed. Additions must be made to the ARRAY.VARIABLE and SWITCH PROC sections of all Phase Two segments on RECORD ONE. Once the compiler in storage (AND RECORD ONE) is completely updated for the new procedure, all its Phase One and Phase Two segments must be restored onto the tapes which comprise the compiler in use. It should be noted that no changes need be made to Phase Three of the compiler in order to insert a new procedure into the system.

Before any additions can be made, however, identifying numbers must be given to the single variables and arrays which are to be read in, to the singles and arrays which are global to the procedure but not read in, and lastly to the procedure name itself. Identifying numbers have to be assigned to these variables because the compiler works with numbers which represent variables, rather than with the variables themselves.

In some instances, in the development of new procedures, variables will be used which are also used in other procedures which have already been incorporated into the system. The code to process these variables can be completely neglected in making changes to the compiler for the new procedure. In no case should they be given a new number, and no new code has to be inserted to handle them.

If a new variable is of the read-in type, it should always be given a number representing a read-in variable. This should be done even if there is the possibility that at some other time it may be used as a variable which is not of the read-in type. Furthermore, if a variable is used in the new procedure as a read-in variable, which was previously used in another procedure as a variable not of the read-in type, then a new number should be assigned to that variable to establish it as a read-in variable. The code which processed it previously for other procedures should now be changed to agree with the new number it has been given.

In order to assign numbers properly to the new variables, a list should be made of all the new variables in the following three categories: single read-in variables, array read-ins, and global single variables and arrays which are not of the read-in type. A check should then be made to find out the current number of the last variable which has been inserted into the compiler from each of these categories. The number of the last procedure in the compiler should also be checked. Then new numbers can be assigned in each category, and to the procedure name, by beginning at the last number which was used in each case.

Suppose for the sake of illustration that the current number of variables and procedures already in the compiler is as follows:

Single Read-In - 63
Array Read-In - 56
Global-Non-Read- 83
Procedures - 85

Then for the example procedure new numbers would be assigned to the lists from each category as follows:

8.3.4

1. Single Variables - Read In - From Comment No. 3 except for those which are already inside the compiler

LOG1 - 64
LOG2 - 65
LOG3 - 66
INT1 - 67
INT2 - 68
INT3 - 69
REL1 - 70
REL2 - 71
REL3 - 72
REL4 - 73

2. Arrays - Read In - From Comment No. 5 except for those already inside the compiler

LOGARY1 - 57
LOGARY2 - 58
RELARY1 - 59
RELARY2 - 60

3. Global Variables - Singles and Arrays - Not Read In - From Comment Nos. 2, 4, & 7 except for those in the two sections above, and those already inside the compiler

INT4 - 84
INT5 - 85
INT6 - 86
INT7 - 87
REL5 - 88
REL6 - 89
INTARY1 - 90
INTARY2 - 91
INTARY3 - 92
RELARY3 - 93
RELARY4 - 94

4. Procedures - From Comment No. 9

PROC.N - 86

Once these numbers have been assigned, the proper changes can be made first to the library, AND RECORD TWO, and then to the compiler, AND RECORD ONE.

Insertion Into the Library

The HYDRO library, AND RECORD TWO, is arranged as illustrated in the following diagram:

ALGOL END CARDS

DECLARATION OF THE SEVEN
DATA READ PROCEDURES

- NEXT.COLUMN
- NUMERICAL.DATA
- ALPHA.NUMERICAL.DATA
- DEFINE.SUBSCRIPTS
- DATA.INPUT
- SINGLE.READ
- ARRAY.READ

FOUR EXECUTION ERROR MESSAGES

DECLARATION OF GLOBAL SINGLE VARIABLES

DUMMY PLOT PROCEDURE

BASE1 CARD

SINGLE.READ CALLS

BASE2 CARD

DECLARATION OF ARRAYS AND
ARRAY.READ CALLS

BASE3 CARD

DECLARATION OF GLOBAL MATERIAL
FOR PROCEDURES

BASE4 CARD

PROCEDURE BODIES

Nothing is ever changed on AND RECORD TWO prior to the DECLARATION OF GLOBAL SINGLE VARIABLES. The code prior to this contains the data reading procedures followed by the data read error messages. These are permanent, and are not altered by additions to the HYDRO system.

The DECLARATION OF GLOBAL SINGLE VARIABLES is the first section to be changed by the addition of a new procedure to the system. These declarations already contain the integer counters, I, J, K, etc., and the dummy real variables, D1, D2, D3, etc. Therefore, nothing has to be done to insert these variables which, in the example procedure, are declared at Comment No. 1. All the other new global single variables, however, located at Comment No. 2, must be declared here. The new integer variables should be added to the integer declaration, the reals to the real declaration, and the logic variables to the logic declaration. The additions which would be made for the example procedure would be as follows. Suppose the old declaration statements were the following:

```
REAL D1, D2, D3, D4, D5. D6, D7, D8, D9,
   MEANVALUE,DEVIATION,CEILING,HISTINCR,VARIATE,ANNUAL,ANNUAL1,
   ANNUAL2,ANNUAL3,ANNUALX,RAINFALLAVERAGE,GRIDAREA,SHEDAREA,
   RAINVOLUME,CONVERTER,REPEAT1,REPEAT2,REPEAT4,REPEAT5.INCREMENT,
   AREA,EXCESSDURATION,PEAKPERCENT,CONST1,TSUBC,
   DURATION2,PERCENTLENGTH,SHAPEFACTOR,MAXABSCISSA,BASELENGTH,
   SYNINCREMENT,SNYAREA,SPECWT,ELEVATION1,DEPTH1,
   YDEPTH,YDEPTH1,MINDEPTH,MAXVALUE,MINVALUE;
INTEGER   I, J, K, L, M, N, P, Q, R, S, T,
   TOTAL1,TOTAL2,TOTAL3,TOTAL4,TOTAL5,TOTAL6,TOTAL7,TOTAL8,
   TOTAL9,TOTAL10,TOTAL11,TOTAL12,TOTAL13,TOTAL14,TOTAL15,
   TOTAL20,TOTAL21,TOTAL22,TOTAL23,TOTAL24,TOTAL25,TOTAL26,
   MAXMIN,TIMEUNIT,INTERVAL,GRIDROWS,GRIDCOLUMNS,
   ISOCHECK,WEIGHTSWITCH,IPRIME,INITIAL2,INITIAL3,
   BEGSTAT,ENDSTAT,MAXPOINTS,MAXAREAS,NODISCH,NOART;
LOGIC   INITIAL1,MESSAGE1,MESSAGE2,
   MESSAGE3,MESSAGE7;
BOOLEAN   PRINTER,PLOTTER,CONTROL1;
```

After the addition of the new single variables these declarations would become:

```
REAL   D1, D2, D3, D4, D5, D6, D7, D8, D9,
    MEANVALUE,DEVIATION,CEILING,HISTINCR,VARIATE,ANNUAL,ANNUAL1,
    ANNUAL2,ANNUAL3,ANNUALX,RAINFALLAVERAGE,GRIDAREA,SHEDAREA,
    RAINVOLUME,CONVERTER,REPEAT1,REPEAT2,REPEAT4,REPEAT5,INCREMENT,
    AREA,EXCESSDURATION,PEAKPERCENT,CONST1,TSUBC,
    DURATION2,PERCENTLENGTH,SHAPEFACTOR,MAXABSCISSA,BASELENGTH,
    SYNINCREMENT,SYNAREA,SPECWT,ELEVATION1,DEPTH1,
    YDEPTH,YDEPTH1,MINDEPTH,MAXVALUE,MINVALUE,
    REL1,REL2,REL3,REL4,REL5,REL6;
 INTEGER   I, J, K, L, M, N, P, Q, R, S, T,
    TOTAL1,TOTAL2,TOTAL3,TOTAL4,TOTAL5,TOTAL6,TOTAL7,TOTAL8,
    TOTAL9,TOTAL10,TOTAL11,TOTAL12,TOTAL13,TOTAL14,TOTAL15,
    TOTAL20,TOTAL21,TOTAL22,TOTAL23,TOTAL24,TOTAL25,TOTAL26,
    MAXMIN,TIMEUNIT,INTERVAL,GRIDROWS,GRIDCOLUMNS,
    ISOCHECK,WEIGHTSWITCH,IPRIME,INITIAL2,INITIAL3,
    BEGSTAT,ENDSTAT,MAXPOINTS,MAXAREAS,NODISCH,NOART,
    INT1,INT2,INT3,INT4,INT5,INT6,INT7;
 LOGIC   INITIAL1,MESSAGE1,MESSAGE2,
    MESSAGE3,MESSAGE7,LOG1,LOG2,LOG3;
 BOOLEAN   PRINTER,PLOTTER,CONTROL1;
```

These additions would be made as part of an AND program to alter and add to AND RECORD TWO.

The next section of AND RECORD TWO to be changed is that following the BASE1 card, which reads THE NUMBER OF THIS CARD IS BASE1. This section contains the calls on SINGLE.READ. A new call on this procedure must be added to this list for each new read-in single variable. These variables are found at Comment No. 3 of the example program. Their SINGLE.READ calls must be placed at the end of the list, just prior to the BASE2 card. A single card should be inserted for each variable, in the order of the numbers assigned to these variables. It is important that these numbers be continuous, i.e. that the Nth card following the BASE1 card be the SINGLE.READ call for single variable N.

These procedure calls consist of the words SINGLE.READ followed by a four number parameter list. The first three parameters consist of the variable name repeated three times, and the last is the number 1, 2, or 3 signifying a real, integer, or alphanumeric variable respectively.

Thus the calls to be added to this list for the new variables of the example are the following:

```
SINGLE.READ(LOG1,LOG1,LOG1,3);
SINGLE.READ(LOG2,LOG2,LOG2,3);
SINGLE.READ(LOG3,LOG3,LOG3,3);
SINGLE.READ(INT1,INT1,INT1,2);
SINGLE.READ(INT2,INT2,INT2,2);
SINGLE.READ(INT3,INT3,INT3,2);
SINGLE.READ(REL1,REL1,REL1,1);
SINGLE.READ(REL2,REL2,REL2,1);
SINGLE.READ(REL3,REL3,REL3,1);
SINGLE.READ(REL4,REL4,REL4,1);
```

The next section of code to be changed on AND RECORD TWO is that following the BASE2 card - the read-in array declarations and ARRAY.READ procedure calls. A new entry, consisting of an array declaration and ARRAY.READ call, should be made for each of the new read in arrays. These are found at Comment No. 5 of the example procedure. These additions should be made at the end of the list, just prior to the BASE3 card. Three cards are entered for each new read-in array. These three-card sets must be entered in the order of the numbers assigned to the arrays. It is important that these sets be continuous, i.e. that the Nth three-card set following the BASE2 card be for the Nth read-in array.

The first card of the set contains the standard declaration of the array. The second and third cards contain the call on ARRAY.READ for the array. This consists of the words ARRAY.READ followed by 8 parameters. The first three parameters consist of the name of the array repeated three times. The fourth is the number 1, 2, or 3 denoting a real, integer, or alphanumeric array respectively. The fifth consists of the first four letters of the array name, surrounded by single quotes. The last three are the three bounds of the array, with zeros supplied as dummies for one and two dimensional arrays. Thus the three-card sets to be added to this list for the new read-in arrays of the example are the following:

```
BEGIN  LOGIC ARRAY LOGARY1 [1:INT1,1:3]  ;
   ARRAY.READ(LOGARY1,LOGARY1,LOGARY1,3,
   'LOGA',INT1,3,0);
BEGIN  LOGIC ARRAY LOGARY2 [1:INT2]  ;
   ARRAY.READ(LOGARY2,LOGARY2,LOGARY2,3,
   'LOGA',INT2,0,0);
BEGIN  REAL ARRAY RELARY1 [1:INT1,1:INT2]  ;
   ARRAY.READ(RELARY1,RELARY1,RELARY1,1,
   'RELA',INT1,INT2,0);
BEGIN  REAL ARRAY RELARY2 [1:INT2]  ;
   ARRAY.READ(RELARY2,RELARY2,RELARY2,1,
   'RELA',INT2,0,0);
```

The next section of code to be added to in the library (AND RECORD TWO) is that following the BASE3 card - the declaration of code which is to be placed global to the new procedure. This material is assembled from three locations in the example program. The first is at Comment No. 4 where some global arrays are declared which are not of the read-in type. In the example, these would be the arrays INTARY1, INTARY2, and RELARY3, which are not read-in by code under Comment No. 5. Secondly, the code at Comment No. 6 which calculates bounds on other global arrays must be assembled as part of the global material. And after this, the code at Comment No. 7 which declares the final global arrays must be placed with the other global material outside of the procedure. This entire block of material - the excess declarations under Comment No. 5, and the code under Comments 6 and 7 - is placed at the end of the sections of code following the BASE3 card, just prior to the BASE4 card. This collection of global code would be the following:

```
COMMENT   GLOBALS FOR PROCEDURE 86;
BEGIN
INTEGER ARRAY  INTARY1[1:INT3] , INTARY2[1:INT2] ;
REAL ARRAY  RELARY3[1:INT3] ;
   INT4 ← INT2/6 ;
   INT5 ←  †(INT1/INT4) + 1 ;
   INT6 ← 0 ;
   FOR I ← 1 STEP 1 UNTIL INT2 DO
      IF RELARY2[I] > REL2  THEN
      INT6 ← INT6 + 1 ;
BEGIN
INTEGER ARRAY  INTARY3[1:INT6,1:INT4] ;
REAL ARRAY  RELARY4[1:INT5,1:INT2] ;
```

The final section of code to be added to the library is the body of the procedure itself. This assumes that the procedure used by PROC.N in the example, i.e. PROC.A, is already in the library. If it were not, the code under Comment No. 8 in the example would have to be added to the library before that of PROC.N was added.

All of the bodies of procedures in the HYDRO library are located after the BASE4 card. PROC.N would be added along with its call to the end of this list. The piece of code to be added would look like the following:

```
COMMENT   BODY OF PROCEDURE 86;
BEGIN
PROCEDURE PROC.N;
   BEGIN
   |
   |
   |(BODY OF PROCEDURE PROC.N)
   |
   |
   END;
PROC.N;
```

This completes the additions to the library for the new procedure PROC.N. Each HYDRO system would have its own method for making these changes, (See Chapter 9 and Appendix I), but at Carnegie Tech they would be made by the submission of a single AND program if INSERT and ALTER statements for

AND RECORD TWO. This would result in a new copy of the library with PROC.N included. This copy, with each line numbered, would then be used to make the necessary changes to the compiler segments on AND RECORD ONE.

8.3.12

Insertion into the Compiler

The compiler on AND RECORD ONE has three Phases, each of which may or may not have more than one segment. At present, in the Carnegie Tech system, Phase One has two segments and Phase Two has four segments. Phase Three occupies only one segment, and will never need more than that. The only reason Phases One and Two occupy several segments is that when merged into single programs they are too large for the capacity of the Carnegie Tech computer. Therefore, these Phases have been subdivided into smaller programs which interconnect with each other and perform the same task as the merged segments would perform.

The compiler segments of Phase One must be altered so that the new procedure and its read-in data variables can be recognized, and appropriate numbers stored for them into the STACK. The segments of Phase Two must be altered so that the bounds of read-in arrays can be checked by the ARRAY.VARIABLE procedure, and code can be executed at the proper label of the PROC switch so that the numbers for the new procedure will be processed when they appear in the STACK.

The segments comprising Phase One are virtually identical. The differences lie in the fact that the calls on PR, SR, and AR for all the procedures and read-in data variables of HYDRO are distributed among the Phase One segments, so that none has the same calls as any of the others, and no call is repeated in more than one segment.

Phase One is altered for the new procedure by adding calls on PR, SR, and AR at the proper labels so that the procedure and its data variables will be recognized when they appear in an input HYDRO program. The procedure and variables to be recognized for the example PROC.N are as follows:

```
PROC.N  - 86

LOG1    - 64
LOG2    - 65
LOG3    - 66
INT1    - 67
INT2    - 68
INT3    - 69
REL1    - 70
REL2    - 71
REL3    - 72
REL4    - 73

LOGARY1 - 57
LOGARY2 - 58
RELARY1 - 59
RELARY2 - 60
```

The calls on PR, SR, and AR to recognize these are as follows:

```
PR(86,'P','R','O','C','.','N','*','*','*','*','*','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*') ;

SR(67,'I','N','T','1','*','*','*','*','*','*','*','*','*','*','*') ;
SR(68,'I','N','T','2','*','*','*','*','*','*','*','*','*','*','*') ;
SR(69,'I','N','T','3','*','*','*','*','*','*','*','*','*','*','*') ;

SR(64,'L','O','G','1','*','*','*','*','*','*','*','*','*','*','*') ;
SR(65,'L','O','G','2','*','*','*','*','*','*','*','*','*','*','*') ;
SR(66,'L','O','G','3','*','*','*','*','*','*','*','*','*','*','*') ;
AR(57,'L','O','G','A','R','Y','1','*','*','*','*','*','*','*','*') ;
AR(58,'L','O','G','A','R','Y','2','*','*','*','*','*','*','*','*') ;

SR(70,'R','E','L','1','*','*','*','*','*','*','*','*','*','*','*') ;
SR(71,'R','E','L','2','*','*','*','*','*','*','*','*','*','*','*') ;
SR(72,'R','E','L','3','*','*','*','*','*','*','*','*','*','*','*') ;
SR(73,'R','E','L','4','*','*','*','*','*','*','*','*','*','*','*') ;
AR(59,'R','E','L','A','R','Y','1','*','*','*','*','*','*','*','*') ;
AR(60,'R','E','L','A','R','Y','2','*','*','*','*','*','*','*','*') ;
```

Notice that SR calls are used for the single variables and AR calls for the arrays. In each case the number of the procedure or variable is the first parameter of the procedure call. This is followed by the letters of the procedure or variable name, each surrounded by quote marks. The remaining parameters are dummies, that is, asterisks, used in order to attain the required number of parameters, 31 for PR and 16 for SR and AR.

These new calls may be added to any segment of Phase One, wherever there is room for them. They do not all have to be inserted into the same Phase One segment. The PR call is inserted after the PP label, that is, the P label for procedures, because the procedure name begins with a P. In like manner, the data are placed after the ID, LD, and RD labels, since they are data variables beginning with I, L, and R. These calls can be placed anywhere at the appropriate labels before, after, or among the other procedure calls located there. These insertions would be made as part of a program to alter AND RECORD ONE, the compiler in storage.

Phase Two can be altered by the same program. Whereas only one segment of Phase One may have been effected by the addition of the new procedure, all segments of Phase Two must be altered in order to accommodate it.

Like Phase One, the segments comprising Phase Two are quite similar. The differences lie in the fact that the processing code at the labels of the PROC switch is different in each. The processing for all of the legal HYDRO procedures is distributed among all the Phase Two segments so that none processes the same procedures as any of the others, and no procedure is processed in more than one segment.

Several things must be changed or added onto in Phase Two. First, since new material has been added to the various sections of the library, RECORD TWO, the numerical values stored into the four BASE values have to be changed in all Phase Two segments. Secondly, additions must be made in all segments to the ARRY switch in the ARRAY.VARIABLE procedure. This must be done in order to test the bounds of all the new read-in arrays. And finally a new addition must be made to the PROC switch in order to create a new label to contain the processing code for the procedure being added. This code may be located in only one of the Phase Two segments; in all others the new PROC switch label will contain a LINK to the segment containing the processing code.

The definition of the four BASE variables is the first thing to be altered on each of the segments of Phase Two. In each segment this code is located immediately following the declaration of the ten Phase Two procedures and the SWITCH PROC. Using example numbers, these variables may have been defined as follows:

```
BASE1 ← 262;
BASE2 ← 327;
BASE3 ← 497;
BASE4 ← 696;
```

Due to the addition of new material to the HYDRO library, the card image numbers of the 4 BASE cards will of course have changed. Their new values can be obtained directly from the print-out of RECORD TWO, or they can be calculated.

The BASE1 card is located in RECORD TWO after the declaration of all GLOBAL SINGLE VARIABLES. For the example, PROC.N, two additional lines or card images were added to this section. Therefore, the BASE1 card would now be at line number 264 instead of 262.

The BASE2 card is located at the end of the section of code containing the SINGLE READS. In the example, ten calls on SINGLE.READ were added to this section. These ten card images, plus the two prior to the BASE1 card, increase the value of BASE2 from 327 to 339.

The BASE3 card is located after the ARRAY READS. Four new array reads were added in the example, each containing three cards, so that twelve new cards were added to this section. Thus, BASE3 would be 497 + 12 + 10 + 2 = 521.

The BASE4 card is located after the GLOBAL DECLARATIONS FOR PROCEDURES. In the example 13 lines were added to this section. Therefore the new value of BASE4 would be 696 + 13 + 12 + 10 + 2 ← 733.

The definitions of the BASE variables should be changed in all the segments of Phase Two. For the example the new values would be as follows:

```
BASE1 ← 264;
BASE2 ← 339;
BASE3 ← 521;
BASE4 ← 733;
```

The ARRAY.VARIABLE procedure is altered next. It is the sixth procedure declared in each segment of Phase Two, and the additions to it are the same for each of the segments. Two parts of the procedure must be changed. New labels must be added to the ARRY switch declaration, and new calls on TEST.BOUNDS must be placed at these labels.

In the example procedure, PROC.N, four arrays are of the read-in type. These, along with their identifying numbers, are as follows:

LOGARY1 [1:INT1,1:3]	57
LOGARY2 [1:INT2]	58
RELARY1 [1:INT1,1:INT2]	59
RELARY2 [1:INT2]	60

The identifying numbers of the bounds involved are:

INT1	67
INT2	68

The first thing to be done to the ARRAY.VARIABLE procedure is to add on to the ARRY switch. Since four arrays are being added, four new switch locations are required. If the old declaration of the switch is the following:

```
SWITCH ARRY ← A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
   A11, A12, A13, A14, A15, A16, A17, A18, A19, A20,
   A21, A22, A23, A23, A25, A26, A27, A28, A29, A30,
   A31, A32, A33, A34, A35, A36, A37, A38, A39, A40,
   A41, A42, A43, A44, A45, A46, A47, A48, A49, A50,
   A51, A52, A53, A54, A55, A56;
```

it should be extended so that it reads as follows:

```
SWITCH ARRY ← A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
   A11, A12, A13, A14, A15, A16, A17, A18, A19, A20,
   A21, A22, A23, A24, A25, A26, A27, A28, A29, A30,
   A31, A32, A33, A34, A35, A36, A37, A38, A39, A40,
   A41, A42, A43, A44, A45, A46, A47, A48, A49, A50,
   A51, A52, A53, A54, A55, A56, A57, A58, A59, A60;
```

Following the additions to the ARRY switch, the new labels declared have to be defined. This is done at the end of the long list of labels already defined after the switch declaration, prior to the line of code labeled L240. The line of code at each ARRY switch label contains a call on the TEST.BOUNDS procedure. The parameters of this call for each of the arrays are the identifying numbers of the bounds of that array. When three variable bounds do not exist for an array, zeros are used where necessary. Thus, for LOGARY1 with the bound INT1, the parameters would be 67,0,0. For RELARY1 with bounds INT1 and INT2, the parameters of the call on TEST.BOUNDS would be 67,68,0. Therefore, the definition of labels for the four new read-in arrays would be:

```
A57:  TEST.BOUNDS (67,0,0) ;
A58:  TEST.BOUNDS (68,0,0) ;
A59:  TEST.BOUNDS (67,68,0) ;
A60:  TEST.BOUNDS (68,0,0) ;
```

Finally, in order to process the procedure, PROC.N, an addition must be made to the PROC switch in all segments of Phase Two. A new label, P86, must be declared there, and then in one of the segments, at the definition of that label, code must be located for processing the procedure. In all other segments a LINK is merely made at the label to get to the segment containing the processing code.

The first thing to be done is to make the addition to the PROC switch. This switch is located after the declaration of the ten procedures in each segment of Phase Two. Only one new switch location will be added, P86. If the old declaration of the switch is the following:

```
SWITCH PROC ← P0, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10,
         P11, P12, P13, P14, P15, P16, P17, P18, P19, P20,
         P21, P22, P23, P24, P25, P26, P27, P28, P29, P30,
         P31, P32, P33, P34, P35, P36, P37, P38, P39, P40,
         P41, P42, P43, P44, P45, P46, P47, P48, P49, P50,
         P51, P52, P53, P54, P55, P56, P57, P58, P59, P60,
         P61, P62, P63, P64, P65, P66, P67, P68, P69, P70,
         P71, P72, P73, P74, P75, P76, P77, P78, P79, P80,
         P81, P82, P83, P84, P85;
```

the new label should be added so that the last line reads as follows:

```
         P81, P82, P83, P84, P85, P86;
```

Following the addition to the PROC switch, the new label declared there must be defined in all segments of Phase Two. The processing code for the new procedure is placed at this label in whatever segment of Phase Two has sufficient room. In all other segments the label is defined by the code:

```
P86:  LINK(N) ;
```

where N would be the number of the segment containing the processing code.

Before the processing code can be designed, certain information is needed. First, a list is required of the identifying numbers of all variables which must be defined (read into or calculated) before the procedure being processed will run correctly. For PROC.N this list is as follows:

Single Variables:

LOG1	64
LOG2	65
LOG3	66
INT1	67
INT2	68
INT3	69
REL1	70
REL2	71
REL3	72
REL4	73

Arrays:

LOGARY1	57
LOGARY2	58
RELARY1	59
RELARY2	60

Then a list should be made of the numbers of all global variables which are defined by the procedure and which will remain defined for use in later procedures. These numbers begin after the last number used for a non-read global variable, which in the example was 83. For PROC.N this list would be as follows:

INT4	84
INT5	85
INT6	86
INT7	87
REL5	88
REL6	89
INTARY1	90
INTARY2	91
INTARY3	92
RELARY3	93
RELARY4	94

Finally the new position on the library, AND RECORD TWO, must be located of the material to be declared global to PROC.N in relation to the BASE3 card. The same must be found for the body of the procedure and its call in relation to the BASE4 card. Suppose the material global to PROC.N lies between card images

BASE3 + 200 and BASE3 + 211

and suppose the body of the procedure and its call lie between card images

BASE4 + 3875 and BASE4 + 3950

With this information known, the Phase Two code for processing PROC.N can be written. This block of code will be given below, and then an explanation will be given for its structure.

```
P86:  PROCESS.VARIABLES ;
      IF ¬ IN.THE.STACK(PROCBASE + 86) THEN BEGIN
         CHECK.STACK(64) ;
         CHECK.STACK(65) ;
         CHECK.STACK(66) ;
         CHECK.STACK(67) ;
         CHECK.STACK(68) ;
         CHECK.STACK(69) ;
         CHECK.STACK(70) ;
         CHECK.STACK(71) ;
         CHECK.STACK(72) ;
         CHECK.STACK(73) ;
         CHECK.STACK(ARRAYBASE +57) ;
         CHECK.STACK(ARRAYBASE +58) ;
         CHECK.STACK(ARRAYBASE +59) ;
         CHECK.STACK(ARRAYBASE +60) ;
         LOAD.STACK(SPECBASE +84) ;
         LOAD.STACK(SPECBASE +85) ;
         LOAD.STACK(SPECBASE +86) ;
         LOAD.STACK(SPECBASE +87) ;
         LOAD.STACK(SPECBASE +88) ;
         LOAD.STACK(SPECBASE +89) ;
         LOAD.STACK(SPECBASE +90) ;
         LOAD.STACK(SPECBASE +91) ;
         LOAD.STACK(SPECBASE +92) ;
         LOAD.STACK(SPECBASE +93) ;
         LOAD.STACK(SPECBASE +94) ;
         IF ¬ IN.THE.STACK(PROCBASE +1)THEN BEGIN
            MOVE.CARD(BASE3 + 2,BASE3 + 5) ;
            MOVE.CARD(BASE4 + 2,BASE4 +14) ;
            LOAD.STACK(PROCBASE +1) ;   END ;
         MOVE.CARD(BASE3 + 200,BASE3 + 211) ;
         MOVE.CARD(BASE4 + 3875,BASE4 + 3950) ;
     END ELSE MOVE.CARD(BASE4 + 3950,BASE4 + 3950) ;
     GO TO L275 ;
```

The first command of this code is a call on the PROCESS.VARIABLES procedure. This call, or in a few cases, a call on the similar PROCESS.VARIABLES.EXCEPT or PROCESS.VARIABLES.UNTIL procedures, will always be the first command of any processing code. It assembles into the ALGOL program being constructed the single variable and array reads which handle all reading of data prior to the execution of the procedure.

The next task to be carried out is to check if the procedure has ever been called and processed before. This check is made by the following line of code:

```
IF ¬ IN.THE.STACK(PROCBASE + 86)THEN BEGIN
```

In the case that the procedure has been called previously in the HYDRO program, PROCBASE + 86 will be IN.THE.STACK, and so only one new line of code has to be assembled into the ALGOL program being constructed for it. That is another call on the procedure. In this case, the entire main body of processing code is skipped, and control passes to the next-to-last line of code which reads:

```
END ELSE  MOVE.CARD(BASE4 + 3950,BASE4 + 3950) ;
```

This makes a call on MOVE.CARD for the card image containing the procedure call. Control then passes back to the label L275 to process the following procedure called in the HYDRO program.

If, on the other hand, the number representing the procedure, PROCBASE + 86, is not found in the STACK, the compiler must carry out all of the processing code for the procedure. This code has three principal parts: calls on the procedure CHECK.STACK, calls on the procedure LOAD.STACK, and finally calls on MOVE.CARD. The first checks if all the necessary data for the correct execution of the program has been entered; the second loads the numbers of new global variables to be calculated by this procedure into the STACK; and the third assembles a set of number pairs representing the procedure and material global to it into the PROGRAM array.

First, the calls on CHECK.STACK are executed. A call must be made on CHECK.STACK for every variable which has to have data read into it in order for the procedure to run properly. If none of these calls on CHECK.STACK leads to an error, then all the variables necessary for the running of the procedure have been defined.

Next, calls are made on the LOAD.STACK procedure. A call is made on LOAD.STACK for each one of the global variables not of the read-in type which will be defined by the procedure.

Once all calls on LOAD.STACK have been made the section of code which calls MOVE.CARD is found. The simple rule to be applied here is that all material has to be moved which will enable the procedure to run correctly. This includes first of all other material which must be global to the procedure body, and then finally the procedure body and its call. For the example procedure, a check is first made to see if Procedure No. 1, i.e. PROC.A, has ever been called and processed. This is done because PROC.A is used by the PROC.N procedure. If this procedure has not been processed, then a call is first made on MOVE.CARD for the material global to it. Following this, a call is made on MOVE.CARD for the body of PROC.A, but not for its call. The call on PROC.A is located inside the PROC.N procedure. Since PROC.A was not previously located in the STACK, but has now been processed, its identifying number should be placed in the stack at this time. Thus, the following call on LOAD.STACK is made:

```
LOAD.STACK(PROCBASE +1) ;
```

This ends the processing of code for Procedure No. 1. This is followed by the assembly of PROC.N.

First, the material which is global specifically to PROC.N is assembled, after which the procedure body and its call are placed. This ends the processing for the procedure. These calls are followed by the two lines of code described previously which assemble only the call for the procedure and return control to L275.

All the sections of code at the PROC switch labels for processing the different procedures follow this same general pattern, though no two of them will be exactly the same. Naturally, in all cases the parameters of the procedure calls on CHECK.STACK, LOAD.STACK, and MOVE.CARD will be different. There will also in general be different numbers of calls each time. The number depends upon how many variables have to be CHECKED, how many have to be STACKED, and what pieces of code have to be assembled by MOVE.CARD. These procedures have been designed in such a way, however, that any situation which may arise can be handled efficiently.

The addition of code at the new PROC switch label completes the changes which must be made to Phase Two.

No additions have to be made to Phase Three in order to compile any new procedure added to the HYDRO system.

Once the compiler in storage has been changed, the new Phase One and Phase Two programs located there must be stored onto the segments which make up the compiler in use. This is done by submitting short programs to load the copies of the new segments from RECORD ONE into the computer core, and run them.

Once the new segments comprising Phase One and Phase Two have been stored, a check should be made to see if PROC.N will run correctly when called in a HYDRO program. Any errors which have been made in altering the HYDRO software to accommodate the new procedure will be readily evident from the output of the ALGOL program resulting from the HYDRO run. These can easily be corrected and the test program checked again.

9. IMPLEMENTATION OF HYDRO ON OTHER COMPUTING FACILITIES

If the HYDRO language is implemented on a different computer than that at Carnegie Tech, some changes will have to be made to the system. It is the purpose of this Chapter to discuss generally what these changes might be, and where difficulties might be encountered in making them.

Several software facilities, available to the Carnegie Tech computer user, have been employed in the development of the HYDRO system. These are described in the first Appendix to this manual. If the reader is not familiar with these facilities, it would help to consult that Appendix before considering the issues discussed in this Chapter.

The major difficulties which would be encountered in implementing HYDRO elsewhere are listed below.

1. Many other computer centers do not have the Control Data G-20 computer, nor will they have the I.B.M. 360 on which HYDRO will eventually be run at Carnegie Tech. Most centers have some smaller model of I.B.M. machine. Even if these centers used the ALGOL language (many of them do not), parts of the HYDRO system would have to be reprogrammed for any computer which differed from the G-20 in word format or length, internal representation of alphanumeric characters, access and availability of secondary tapes and discs, smaller amount of core storage, and different size print line. Differences in the two machine codes would also make small alterations necessary in the system.

2. On many other computers, FORTRAN or some other algebraic language is used in the place of ALGOL. If this were the case, the entire

compiler (and all subroutines, if HYDRO was being copied) would have to be transformed into the FORTRAN language. In general, however, no difficulty would be involved here besides the time and effort, since no capabilities available in ALGOL-60 are utilized in the HYDRO system beyond what is also available in FORTRAN and most other algebraic languages.

3. There are many versions of ALGOL besides ALGOL-60 (one of these being Carnegie Tech's ALGOL-20) so that even if ALGOL were available at the copying center, it would probably be different in several ways from Carnegie Tech's ALGOL. The differences between the ALGOL-20 of Carnegie Tech and the international ALGOL-60 are outlined in Appendix I; differences similar to these would likewise exist between ALGOL-60 and the ALGOL of any other computation center. These differences involve such things as symbols used for operators, brackets and quotes; available standard functions and the way they operate; reserved identifiers; logic variables; labels; and especially input/output formats. Wherever differences such as these did exist, alterations would have to be made to the HYDRO compiler in order that it run at the new center.

4. Differences undoubtedly would also exist between the translators of the two centers, even if both used the ALGOL language. Where these differences had an effect on the way code was interpreted or executed, changes would again have to be made in the HYDRO system.

5. Phase Three of the HYDRO compiler, and to a small extent Phase One, makes use of the ALGOL relocatable library routines (ANDFILE, DISCREAD, and DISCWRITE) available in the Carnegie Tech ALGOL language.

These routines make possible the transfer of code between core and tape/disc. Obviously, no other center would have these same routines, but most centers would in some way make provision for moving blocks of code from one storage component to another. To the degree that these routines differed from those at Tech, Phases One and Three of the compiler would have to be altered. Furthermore, Phase Three uses the AND SCRATCH AREA on disc for construction and temporary storage of the ALGOL program which represents the input HYDRO code. This program cannot be constructed in core due to the presence of Phase Three operating there. At another center, where a large construction and storage area such as the AND SCRATCH AREA did not exist, some other area, perhaps on tape, would have to be utilized for this purpose.

6. The HYDRO compiler is divided into segments because it is too large to fit into core all at one time. A similar problem would exist at most other centers, and provision would have to be made for it. At Tech, the segments of the compiler are stored on special areas of the disc, and the relocatable library procedure, LINK,is used to pull these segments into core one at a time for running. OWN variables are used as a means of keeping variables declared and defined during a transfer of segments. At any other computation center, the memory size may differ from that at Tech. Furthermore, a different type of secondary storage may be employed than that at Tech, and OWN type variables may not be available. There may be no way, such as LINKing, of altering the contents of core during the execution of a single program. All of these problems would necessitate changes in the HYDRO system.

7. Another software facility available at Carnegie Tech and widely used in HYDRO is the AND system. As was stated previously, the AND SCRATCH AREA is used for the storage of input data and the construction of the final ALGOL program. Furthermore, the AND records are used for the storage of the language and a copy of the compiler, and are drawn from extensively during Phase Three of the compilation process. The AND language itself is used to edit and implement the HYDRO language when new additions are made to it. In order to run the HYDRO language at another center, some sort of records would have to be available analogous to the AND records, with a means of drawing any amount of code from any part of these records. A means would also have to be available, such as the card image numbering system of AND, of identifying the position of information at any point in these records.

In conclusion it should be noted that some of these differences would present major difficulties, and others only lesser ones. The fact that a computer is a different model than the G-20 (point 1), or that it uses the FORTRAN language (point 2), or that it has a different version of ALGOL or a different ALGOL translator than that at Carnegie Tech (points 3 and 4), does not present a major block to the implementation of HYDRO. Reprogramming of minor difficulty is all that is involved in overcoming these differences. However, the storage size of a particular computer, as well as its peripheral software equipment (points 5, 6, and 7), are of major importance. A great amount of storage is necessary because the HYDRO compiler is so long, and because its length increases with each new addition to the language. Furthermore, the functioning of HYDRO is highly dependent upon the use of segments, relocatable library routines, and the AND system. Where similar facilities to these did not exist, major difficulty would be involved in implementing HYDRO.

BIBLIOGRAPHY

1. Naur, P., editor. "Revised Report on the Algorithmic Language ALGOL-60." *Communications of the ACM*, Vol. 6, January 1963, 1-17.

2. McCracken, D. D. *A Guide to ALGOL Programming*. New York: John Wiley and Sons, 1962

3. Fierst, J. W., editor. *ALGOL-20 Language Manual*. Carnegie Institute of Technology, 1965.

4. -----. *User's Manual*. Carnegie Institute of Technology Computation Center, 1965.

5. -----. *G-20 Central Processor/Machine Language Manual*. Control Data Corporation, Publication No. 611, 1965.

6. Moore, J. A. *WHAT*. Carnegie Institute of Technology, Computation Center, 1965.

7. -----. *THAT*, A Language Manual. Carnegie Institute of Technology, Computation Center, 1965.

8. McNally, W. D. *HYDRO Users Manual*. Carnegie Institute of Technology, Civil Engineering Department, 1966.

APPENDIX I

RELATION OF HYDRO TO CARNEGIE TECH COMPUTING FACILITIES

This Appendix is intended for use by readers unfamiliar with the software computing facilities of the Carnegie Tech Computation Center. Sections A, B, and C of the Appendix explain various aspects of the ALGOL language used by the HYDRO system. The differences between ALGOL-20, the Carnegie Tech version of ALGOL, and ALGOL-60, the international reference language, are outlined in Section A. The next two sections of the Appendix B and C, explain the input-output conventions of Carnegie Tech ALGOL, that is, formatted read and print statements.

Section D of the Appendix gives a detailed description of the ALGOL Relocatable Library routines, LINK, ANDFILE, DISCREAD, and DISCWRITE. These are used by the HYDRO compiler to link back and forth among the compiler segments and to transfer information to and from tape or disc files.

The next section, E, explains the assignment, location, calling, and function of the segments on which the compiler is stored.

Section F describes the AND system used by HYDRO both for compiling an ALGOL program from HYDRO code, and for storage and alteration of the HYDRO compiler and language systems. Typical AND commands are also explained.

The final section of the Appendix, G, describes the WHAT language used briefly in Phase Three of the HYDRO compiler, as well as the bar routines used in that WHAT program.

More information on the material of Sections A, B, C, D, and E of this Appendix can be obtained from the ALGOL-20 Language Manual (3), published by the Carnegie Tech Computation Center. The Users Manual (4), published by the same Center, gives additional material concerning Section F. Finally, the WHAT Language Manual (6) as well as reference (7), will further explain the material in Section G of the Appendix.

A. ALGOL-20 vs. ALGOL-60

ALGOL-20 is simply the ALGOL-60 language modified somewhat to operate on the Carnegie Tech computing system. The ALGOL-20 language has been employed for programming throughout the HYDRO system, in both the compiler and language. This ALGOL is quite similar to that outlined in the Revised ALGOL-60 Report (1); the differences between the two versions will be discussed below.

The first difference is that ALGOL-60 makes no provision for input/output. A formatted language has been developed at Tech for ALGOL input/output and is discussed in Sections B and C of this Appendix.

The ALGOL of Carnegie Tech also has relocatable library subroutines which are employed by the HYDRO system for the transferral of code between various areas of computer storage. These are discussed in Section D of this Appendix.

A third area where differences occur is that of the symbols used to designate certain operations. The following short list illustrates differences between ALGOL-60 and ALGOL-20 for the programmed material covered by this manual. In certain cases two adjacent symbols must be used for the representation of a single symbol of ALGOL-60.

ALGOL-60	ALGOL-20
:=	←
×	*
÷	↓(/)
≤	¬>
≥	¬<
‘’	' '

Throughout the HYDRO system the left arrow is used as the replacement operator, and the single apostrophe is used to represent string quotes. Truncation (↓) is also permitted in ALGOL-20, and is used with the division sign (/) to achieve the same result as the division operator (÷) of ALGOL-60.

All of the ALGOL-60 standard functions, plus others developed at Carnegie Tech, are available in ALGOL-20. Of these ABS, SIN, COS, EXP, HALT, and SQRT are used by the present version of HYDRO.

All of the reserved identifiers of ALGOL-60 are also reserved in ALGOL-20. However, ALGOL-20 has several reserved identifiers not found in ALGOL-60. These are as follows: FORWARD, HALF, INDEX, INPUT, LIBRARY, LOGIC, MAX, MIN, MOD, MONITOR, NAME, OUTPUT, PRINT, PUNCH, and READ. Of these LIBRARY, LOGIC, NAME, PRINT, and READ are used by HYDRO, and will be explained later.

Variables of type HALF, LOGIC, and INDEX are allowed in ALGOL-20 besides the normal INTEGER, REAL, and BOOLEAN variables of ALGOL-60. Of these types, only LOGIC, INTEGER, REAL, and BOOLEAN are used by HYDRO.

Other differences between ALGOL-20 and ALGOL-60 are as follows: Identifiers may not contain spaces in ALGOL-20; periods may be used instead. ALGOL-20 cannot distinguish between left and right string quotes, since the apostrophe is used to represent both. Therefore, strings may not contain strings in ALGOL-20. Finally, a label should be an identifier in ALGOL-20, never an unsigned integer.

Besides these, there are other differences between ALGOL-60 and ALGOL-20 which are of a minor nature. Most of these have no effect whatsoever on the HYDRO system. A full description of all of these differences can be obtained from the Carnegie Tech ALGOL Language Manual (3).

B. FORMAT READ

The ALGOL-60 Report (1) makes no provision for input/output. The input convention partially descirbed here was designed and is in use at Carnegie Tech, and is described fully in the ALGOL Language Manual.(3).

There are two places in the HYDRO system where formatted read is used, and it is used in a similar way in both instances. In the NEXT. COLUMN procedure of the compiler's Phase One, NAME and READ statements are used to copy the card images one by one of the input HYDRO program into the IMAGE vector for processing. In the similar NEXT.COLUMN procedure of the data read procedures, NAME and READ statements are used to copy data card images one by one into the CHAR vector for reading. There is no formatted data reading in any of the ALGOL procedures which are assembled from HYDRO procedure calls. All the reading for these ALGOL procedures is done by the special data reading procedures assembled at the head of the ALGOL program. These data reading procedures have been explained in Chapters 6 and 7 of this manual.

In both instances of formatted read in HYDRO, NAME and READ statements are used. The NAME statement tells the computer what variables are to be read into, and the READ statement indicates the format in which they will be read.

The formatted read of Phase 1 is as follows:

```
NAME(I → 80 (IMAGE [I])) ;
READ(<W>, → 80 <1A>) ;
```

That of the data read procedures is very similar:

```
NAME(II → 80(CHAR [II])) ;
READ(<E>, → 80 <1A>) ;
```

The two NAME statements say the following. Eighty units of the IMAGE vector are going to be successively read into, from IMAGE [1] to IMAGE [80]. Eighty units of CHAR are to be read into, from CHAR [1] to CHAR [80]. The variables I and II are iteration variables and are not of importance. In the first READ statement the letter "W" tells the computer to read a single data card, printing it at the same time on the user's output. In the second READ statement, the letter"E" tells the computer to read a card but not to print it. The expression " → 80" says to read 80 pieces of data from the card into the variable locations NAMED above according to the format designated within the format brackets (< >) which follow the number 80. The format designation "1A" tells the computer that one character at a time is to be stored alphanumerically into the successive positions of the IMAGE and CHAR arrays. Therefore, it can be seen that these NAME and READ statements simply make copies of the 80 columns of the data cards onto the logic IMAGE and CHAR arrays.

Many other types of READ statements exist in Carnegie Tech ALGOL besides the simple ones illustrated above. However, these are not used in the HYDRO system and will not be discussed here. More information on formatted reading in ALGOL can be obtained from the ALGOL Language Manual (3).

C. FORMAT PRINT

The output conventions presented here were developed at Carnegie Tech for use with the ALGOL-60 language. Formatted printing is similar in many respects to formatted reading, and is described fully in the ALGOL Language Manual (3).

There are no occurrences of the use of formatted printing in the HYDRO compiler. However, it is widely used in the HYDRO "language," that is, in the ALGOL procedures assembled from HYDRO input code. Almost every one of these procedures employs formatted printing in some form for the output of computed results to the user.

Several examples of formatted printing are discussed below to enable the reader unfamiliar with these conventions to understand their use in the HYDRO procedures. The first and simplest example of formatted printing is its use to print a single line of information, usually a title, on the user's output. An example of this might be as follows:

```
PRINT(< 1C,33B,'AVERAGE RAINFALL OVER THE AREA',E >);
```

The code within the format brackets (< >) is called a "format program." When executed it causes a line to be printed as follows: The column pointer on the print line is set to column 1 by the 1C instruction. Then 33 blanks (33B) are put into the print line, moving the column pointer to position 34. At this point the text within the string quotes ('---') is entered into the print line. The letter E then says to execute, that is, print what is in the print line. The result is the title

```
AVERAGE RAINFALL OVER THE AREA
```

beginning in column 34 of the 120-column printed page.

A second example is an instruction to PRINT a row of column headings across the page. The code to achieve this might look like the following:

```
PRINT(< 10C,'CENTER',5B,'ZONE',5B,'ISOHYET',5B,'NETAREA',3E >) ;
```

This code would set the column pointer to column 10, and at that point enter the word CENTER into the print buffer. Then five blanks would be added, and the word ZONE entered. In like manner the words ISOHYET and NETAREA would be entered, with 5 spaces between each. The command 3E would print the resulting line and skip two lines after it. The result would look like the following, beginning in column 10, with 5 spaces between each word:

```
CENTER     ZONE     ISOHYET     NETAREA
```

The principal purpose of printing titles and column headings is so that numerical data can eventually be printed beneath them. When numerical data are printed, a NAME statement must be used along with the PRINT statement. The NAME statement specifies which variables are to be printed, and the PRINT statement the format in which this is to be done. An example might be the following:

```
NAME(I,J,SERIES[I,J], DURATION[I]  ) ;
PRINT(< 36C, 2D, 3B, 2D, 10B, -4Z.1Z, 10B, -3D.4D, 2E >) ;
```

These two lines would cause the four values of the variables NAMED to be printed across a line in the format specified by the PRINT statement. That is, nothing would be printed until column 36, (36C). At that point, the value of I would be printed in format 2D, i.e. 2 digits, leading zeros not included. If "I" was a two digit number, both would be printed. If it was a single integer, only one digit would be printed. Then, there would be 3 blank spaces, followed by J printed in 2D format.

Then 10 blanks would appear before the printing of the value in SERIES [I,J]. This would be printed in format -4Z.1Z, meaning 4 characters to the left of the decimal point with leading zeros added if necessary, and preceded by a minus sign if the value is negative. Then the decimal point would come, followed by one digit, a zero if necessary. This would be followed by 10 blank columns after which the value of DURATION [I] would be printed in format -3D.4D. This means DURATION [I] would be printed with a minus sign, only if negative, a maximum of 3 digits to the left of the decimal point with no leading zeros added, and a maximum of 4 digits to the right with no trailing zeros addad. The line would be printed by the command 2E, which would both print the line and skip a line after it. Thus, if the values of the four variables were as follows:

```
               I  = 15
               J  =  2
   SERIES [15,2]  = -25
 DURATION [15]    = 82.67
```

the printed line would appear as below, beginning in column 36 of the printed output page:

```
15    2          -0025.0              82.67
```

A further complexity which can be achieved with the use of formatted printing is the mixing of titles and numbers in the same print line. To achieve this, the variables to be printed are first NAMED as before, and then the PRINT command specifies both the printed titles and the format for printing the numbers. An example of this might be as follows:

```
 NAME(I, SERIES [I]) ;
PRINT(< 50C, 'NUMBER', 3D, ' = ', 4D.2Z, E >) ;
```

The two variables, I and SERIES [I], are accessed for printing by the NAME statement. Then the print line is set up for printing as follows: Nothing is placed in the first 49 columns. The word NUMBER is placed into the 6 columns beginning at column 50. Then the value of the variable I is inserted in format 3D, a maximum of 3 digits for the integer with no sign and no decimal point. Following this, an equal sign is inserted with a space before and after it. Then the value of SERIES [I] is inserted in format 4D.2Z, that is, a maximum of 4 leading digits with no leading zeros added, and 2 trailing digits with zeros added if necessary. Finally the resulting print line, set up in the print buffer, is printed by means of the execution symbol, E. No lines are skipped afterwards because only 1E was specified. Thus if I and SERIES [I] had the following values at the time of execution of the above NAME and READ statements:

```
          I  = 15
SERIES [15]  = 673.2
```

the printed result would be as follows, beginning in column 50:

```
NUMBER 15  =  673.20
```

Further complexity than this can be built into a print line by the use of iteration statements and so-called "replicators." The following will illustrate this concept.

```
FOR I ← 1 STEP 1 UNTIL 10 DO BEGIN
   NAME(AREA [I] , J → TOTAL(MAXCURVE [I,J] )) ;
   PRINT( < 20C, 5D, 12B >, → TOTAL <2D.1Z, 1B >,< E > ) ;  END ;
```

In this example, 10 lines of output will be printed, one for each value of I. In each of these lines TOTAL+1 numbers will be printed -- the AREA, and TOTAL values of the MAXCURVE. The section of the NAME statement reading J → TOTAL(MAXCURVE [I,J]) has the same effect as

```
FOR J ← 1 STEP 1 UNTIL TOTAL DO
  NAME(MAXCURVE [I,J]) ;
```

Likewise the section of the print statement reading →TOTAL< 2D.1Z, 1B > has the same effect as the following, if TOTAL = 3:

```
< 2D.1Z, 1B, 2D.1Z, 1B, 2D.1Z, 1B >
```

Thus, if for I = 1 and TOTAL = 3 the values of the variables are as follows:

```
AREA [I]  = 20541
MAXCURVE [1,1]  = 95.6
MAXCURVE [1,2]  = 42.8
MAXCURVE [1,3]  =  8.7
```

Then the printed line would begin in column 20 and look like

```
20541              95.6 42.8  8.7
```

Because I is iterated to 10 there would be 10 lines like this, one following another. A total of 10 AREA and 30 MAXCURVE values would be printed.

The above 5 examples illustrate most of the printing conventions employed within the HYDRO procedures. These are, of course, combined in various ways, but a study of these examples should be sufficient to explain most of the printing done in HYDRO.

D. ALGOL LIBRARY ROUTINES

The ALGOL relocatable library routines of LINK, ANDFILE, DISCREAD, and DISCWRITE, used by the compiler, are described briefly in Chapter Four of this manual. A more detailed description of these procedures, given below, is taken from the ALGOL Language Manual (3).

Relocatable library routines are coded in machine language, and may be accessed by any user and loaded into any ALGOL program. Relocatable routines must be declared as "library procedures" at the head of the block in which they are used. These declarations in HYDRO are similar to the following:

LIBRARY PROCEDURE LINK, DISCWRITE;

depending upon which procedures are declared in each of the HYDRO compiler Phases.

The procedure LINK (N) loads the program on Segment N from disc storage into core and enters it at the first BEGIN. If the calling program and the called segment have identical declarations of OWN scalars and OWN arrays, none of these OWN variables will be disturbed. Executing LINK causes all non-OWN variables to become undefined. Control transfers automatically to the outermost BEGIN of the newly loaded segment, that is, the first statement executed is always the first statement of the segment program. Alarms are given if the parameter to the LINK call is less than 1, or if segment N was not previously dumped into storage as an ALGOL segment.

Before explaining the ANDFILE, DISCREAD, and DISCWRITE routines, an explanation should be given for "files." Disc and tape storage space at Carnegie Tech is divided into sections called "files." Files are recorded on disc or tape or extracted from disc or tape in units of 320 4-character words, called "blocks." Each file is assigned space on disc or tape in units of complete blocks, i.e. in multiples of 320 computer words. Thus, when either DISCREAD or DISCWRITE is used in the HYDRO compiler, 1600 words are transferred at a time. This is a multiple of 5 times 320, and is thus an even number of blocks.

The ALGOL disc/tape routines, i.e. DISCREAD and DISCWRITE refer to particular files by an integer value between 1 and 19. This is called the "logical file type." Some logical file types have been pre-assigned in certain fixed files in the Tech system. One of these, the AND SCRATCH AREA, which is written onto by the HYDRO compiler, is logical file type one. The file types 10 through 19 have not been pre-assigned, and can be assigned to any file the user wishes by means of the ANDFILE routine. These files can then be accessed and written upon by the other library read/write routines simply by referring to their logical file number.

The relocatable library routine ANDFILE allows an ALGOL program to assign a logical file number to any AND file, that is, to an AND program belonging to a particular user. ANDFILE is called with the following parameters:

```
ANDFILE(USER,PROG,LFTYPE,BUFFLOC,ERREXIT) ;
```

where:

USER = the identifying number of the person whose file is to be numbered. This is an eight-character alphanumeric string surrounded by string quotes. In HYDRO this string is 'CI35WM01'.

PROG = the number of the user's program or file which is to be assigned a logical file number.

LFTYPE = the logical file number to be associated with this AND file. $1 \leq \text{LFTYPE} \leq 19$.

BUFFLOC = an array element which is the first element of an available declared vector of at least 320 words.

ERREXIT = a label to which ANDFILE will exit if any of its parameters are improper, or if the designated AND file does not exist.

The effect of calling ANDFILE is to look up the AND file or record in the AND Directory of records, and to assign to it the logical file type, LFTYPE. The 320 locations starting at BUFFLOC are used as a buffer to read the AND Directory. After the call on ANDFILE this space is available for other uses.

The DISCREAD library routine reads information from disc or tape, (i.e. from an AND file) into an array in core memory. The DISCWRITE routine will write the contents of an array in core memory out onto a file on disc or tape. These routines allow the user to move binary information in bulk between core memory and disc/tape files. Their parameters are as follows:

```
DISCREAD(NWDS,FIRSTLOC,LFTYPE,BLKNO,ERREXIT) ;
DISCWRITE(NWDS,FIRSTLOC,LFTYPE,BLKNO,ERREXIT) ;
```

where

NWDS = number of machine words to be read or written.

FIRSTLOC = an array element which is the ALGOL name of the first word in core memory to read into or be written from.

LFTYPE = number assigned to the logical file to be read from or written onto.

BLKNO = number of the first 320-word block to be read or written onto.

ERREXIT = a label to which DISCREAD or DISCWRITE will exit if any parameters are improper, or if reading or writing goes beyond the end of a file.

Starting at the block number BLKNO of the tape or disc file with logical file type LFTYPE, these routines will read or write as namy blocks as are needed for the number of words NWDS. DISCREAD reads into core memory exactly NWDS words, where NWDS is not necessarily a multiple of 320. DISCWRITE, on the other hand, will write extra meaningless words if need be to complete the last 320-word record.

E. SEGMENTS

The HYDRO compiler is an ALGOL program which is too long to fit into core memory at one time. Therefore, it is divided into subportions, called segments. Each segment is stored onto disc or tape after it is compiled and is then called into core memory as needed. Seven segments are used presently by the HYDRO compiler.

Each ALGOL-20 program which is to be stored onto a tape segment must contain in its outer block a system statement of the form

```
SY SEGMENT N
```

The variable N specifies the segment number on which the program is to be stored. Each segment on disc/tape is 10,240 machine words long. If a segment program contains more words than this, it must be stored onto successive segment areas.

Each segment program must be a complete ALGOL-20 program, that is, it must have a set of matching BEGINs and ENDs, as well as declarations of all identifiers used within the segment.

After a program which is to be stored onto a segment has been compiled (i.e. after it is submitted for running, and the END which matches the first BEGIN is found), the code which has been compiled and any relocated subroutines will be stored onto disc or tape at the segment location specified by the SY SEGMENT command. ALGOL will then continue reading cards, expecting to find another program, which may also be a segment for storing. Compilation terminates after a program which is not a segment is processed; this program is then run. This is the reason a short dummy program is placed at the end of each of the segments on AND RECORD ONE. These dummy programs are as follows:

```
BEGIN
PRINT  ( < 'THE SEGMENT IS STORED', E > ) ;
END ;
```

The statement, SY SEGMENT N, in an ALGOL program to be stored onto a segment is a "system statement." The SY of the statement must be punched in card columns 1 and 2 with the remainder of the statement lying between column 4 and the right margin. System statements are instructions in an ALGOL-20 program which inform the ALGOL translator to modify certain aspects of the translation process. That is, system statements are executed by the translator at compile time rather than by the object program at execution time. System statements are executed as they are encountered by the translator as it scans once through the ALGOL source program and take effect immediately thereafter. Thus, the SY SEGMENT statement in each of the compiler Phases has no effect when the Phase is LINKed to during a HYDRO program.

It is necessary in the HYDRO compiler to be able to retain data in core memory between the operation of different segments. However, linking from one segment ot another involves exiting from the outermost block of the current segment, thus making all variables declared in the segment undefined. To overcome this, the same block of storage locations is used in all segments for OWN scalars and arrays. Thus, if identical declarations of OWN scalars and arrays are made in different segments, as is done in the HYDRO compiler, all these values may be transmitted from one segment to another. All other scalars and arrays are undefined when a new segment is loaded into core for running.

F. THE AND SYSTEM

The AND language provides a means of storing onto and updating the HYDRO records referred to in this manual as AND RECORD ONE and AND RECORD TWO. This is done when changes or new additions are made to the HYDRO language. Also, the AND SCRATCH AREA, a logical file available for use by any AND program, is used by the HYDRO compiler for the assembly of the ALGOL program representing the input HYDRO code. A full description of the AND (AlphaNumeric Director) system, its language, and the areas of the computer involved, is given in the Carnegie Tech Users Manual (4). Only those parts of the language which are involved in augmenting the HYDRO language will be discussed here.

The AND system is useful both as a general librarian for storing files of alphanumeric text, and as a text editor for altering that information. When using AND, long programs do not have to be submitted more than once. After being stored into the AND system they can be accessed, altered, and added onto at will. This is done by means of AND programs which contain only AND instruction cards and new text. Making use of the AND system, the HYDRO compiler can be altered easily and restored onto its segments, and the language can be accessed easily during the running of HYDRO programs.

AND stores and modifies information in groups of 80 characters, that is, it works with card images. It treats each card image as a unit, and does not examine its contents. Any information can be treated as text by AND.

AND stores text in the disc memory in logical groups called "records" or "files." Each AND record is filed under the usage number of the programmer who created it. There may be many different AND records filed under any usage number, so each is distinguished by a "file number" assigned by the programmer when he first creates the record. Thus for HYDRO there are two AND FILES or AND RECORDS, ONE and TWO, for the user CI35WM01.

AND maintains a master Directory which is used to locate current AND records. The Directory entry for each AND record contains the number pair: Usage Number, File Number. A particular record is selected for a user by means of his user number on the AND program job card plus the AND command "FILE" followed by a file number. If a user wishes to access another person's AND record he must use the command "USER" along with "FILE."

AND is called into operation by a job card which contains "AND" as the system name, followed by cards containing AND instructions and cards containing text.

A programmer may use AND to take card images from any existing AND record; however, AND will not allow a programmer to create new AND records except under his own number. Thus, no user may change in any way the contents of anyone else's AND records.

AND uses a temporary work area on disc called the AND SCRATCH AREA. Under control of the instruction cards, AND collates into the SCRATCH AREA any desired combination of card images from existing AND records with card images of new text. The resulting text can be copied from the SCRATCH AREA back onto AND records on disc by a "DUMP" instruction. The dumped text may replace an existing record or may be assigned a new program number and become a new AND record.

There is a "dump count" associated with each AND record. Each time a DUMP instruction writes information onto an AND record, the associated dump count is incremented by one. To alter any record, the user must specify the current value of this dump count along with the file number associated with the record.

Every card on any particular AND record is numbered. The numbers for any record begin at one and increase consecutively up to the number of card images currently contained in the record. These numbers appear to the right of AND text when it is printed from the AND SCRATCH AREA after a collation there. Thus, when editing an AND record the user refers to card images by numbers which he obtains from his latest AND program output for that particular record.

There are relatively few AND instructions which a user must know in order to alter or add to the HYDRO system. The initial ones are the USER and FILE instructions. These select a record to be edited. The USER instruction consists of the word USER followed by a user number. The FILE instruction consists of the word FILE followed by the AND record or file number, a slash (/), and the dump count of that file. Thus, to select AND RECORD ONE of USER CI35WM01, with a dump count of 15, the commands would be:

```
AN USER CI35WM01
   FILE 1/15
```

The FILE instruction sets a pointer on the AND record to the first card image of the record.

The first editing instruction which could be used in INSERT AFTER. This consists of the words INSERT AFTER followed by the number of a card image on the AND RECORD which is to have text inserted after it. That text must follow this command in the AND program. The INSERT AFTER command does two things. It first moves into the SCRATCH AREA all code on the AND record from the card image at which the AND pointer is currently located down to the card image named in the INSERT AFTER command. It leaves the pointer at the card image after that. It then inserts after this the text in the AND program which follows the INSERT AFTER command down to the next AND instruction. Thus if the AND pointer was at card image 1 and the user wished some code inserted after card image 250 he would use the command

```
AN INSERT AFTER 250
```

which would load card images 1 to 250 into the SCRATCH AREA and insert the code which followed this command after these card images on the SCRATCH AREA.

Another text editing command is the ALTER command. If the user only wishes one card altered, he uses the word ALTER followed by the number of that card. If he wishes more than one consecutive card altered, he uses the word ALTER followed by the number pair (separated by the word TO) of the first and last cards he wishes altered. Examples of these two cases are as follows:

```
AN ALTER 250
```

or

```
AN ALTER 250 TO 275
```

Any number of consecutive cards can be altered that the user wishes. The ALTER command must be followed by the text the user wishes substituted for the cards he is altering. Like INSERT AFTER, ALTER does two things. It assembles into the SCRATCH AREA the code on the AND record from the pointer down to but not including the card image number following the word ALTER; it then sets the pointer to the card image following that of the last number given in the ALTER command. After this code is assembled, it loads into the SCRATCH AREA the text following the ALTER command in the AND program.

The user can also delete material on the AND record without substituting anything in its place. For this the DELETE command is used. This functions in a similar way to the ALTER command. Example DELETE commands might be

```
                    AN DELETE 250
or                  AN DELETE 250 TO 275
```

These move the AND pointer down and assemble material into the SCRATCH AREA just as the ALTER command does, but these commands do not follow this material with any new material. Therefore, another AND instruction always follows an AND DELETE instruction in the AND program.

If the user wishes only to load certain material from the AND record to the SCRATCH AREA, he uses the LOAD command. This would have the form

```
                    AND LOAD 250
or                  AND LOAD 250 TO 275
```

This instruction moves the pointer on the AND record to the card image of the first number following the word LOAD. It then loads into the

SCRATCH AREA the card images designated, moving the pointer down accordingly.

Once a new program is assembled fully on the SCRATCH AREA, a printing of its contents along with the new card image numbers associated with the text can be obtained by the command

```
AN PRINT SCRATCH
```

A new record created on the SCRATCH AREA is dumped onto the AND record to replace the previous contents there by the command

```
AN DUMP
```

And an AND program is ended by the command

```
AN DONE
```

An AND program may also be ended by calling for a RUN of the program assembled on the SCRATCH AREA. The command to accomplish this consists of the word RUN, followed by the system name (ALGOL in this case), followed by the word TAPE. Thus the command

```
AN RUN, ALGOL, TAPE
```

would cause the ALGOL program assembled on the SCRATCH AREA to be loaded into core and executed.

These commands are all that are required for the developer of HYDRO to use in altering and expanding the HYDRO compiler and language in the Carnegie Tech system.

There are many other AND instructions which can be used to achieve different ends than those outlined above. These, along with the commands above, are explained in detail in the Carnegie Tech Computation Center Users Manual (4).

G. THE WHAT LANGUAGE AND BAR ROUTINES

In the final lines of Phase Three, the HYDRO compiler makes use of a short 6-line program in the WHAT language to run the ALGOL program, assembled by Phase Three onto the SCRATCH AREA. This program is as follows, with appropriate columns marked for the beginning of significant fields:

```
Col              Col       Col  Col
 1                15        20   24
 ↓               ↓          ↓    ↓
 WH              CLS        0    1                     Line 1
                 TRM             |15                   Line 2
                 TRA             A + 2                 Line 3
                 NAM             ALGOL                 Line 4
                 CAL             A - 1                 Line 5
                 TRM             |23                   Line 6
```

A full understanding of this short routine involves first of all an understanding of the fundamentals of machine coding; secondly, of the WHAT language and its relation to machine code; and finally, of the bar routines available for use with the WHAT language. There are several manuals (5, 6, and 7) explaining all of these systems in detail, and of course, all of this explanation cannot be given here. This section of the Appendix will only strive to give the reader an understanding of the short program above. In explaining it each line will be referred to by line number.

The WHAT language is designed to permit the use of machine code within an ALGOL program, in order to achieve capabilities unavailable in ALGOL alone. During compilation when the ALGOL compiler reads a card with "WH" in the language field, control is transferred out of the ALGOL compiler to the WHAT assembler. The assembler translates cards containing WHAT code until a card is read with "AL" in the language field. This returns control to the ALGOL compiler. The inclusion of WHAT code within an ALGOL program has no effect on the ALGOL translator.

In Line 1 of the WHAT program above, a machine language opcode (CLS) is used. This represents "clear and subtract." It is done in mode zero, with 1 as the operand. This results in storing -1 into the accumulator of the computer in preparation for the next command which involves a bar routine.

Line 2 contains the command to "transfer and mark" (TRM) to bar routine fifteen (|15), that is, go to the bar-fifteen routine and execute it and then return to this position in the WHAT program. The bar-routines are nothing but pre-programmed routines, switches, and definitions available from the monitor during the running of a WHAT program. The bar-fifteen routine has as its function the disignation to the monitor of the subsequent source of code, whether it be from cards or from tape/disc. With the accumulator set to -1, a call on the |15 routine establishes the beginning of the AND SCRATCH AREA as the source of subsequent card images. This is what is desired in HYDRO since the ALGOL program to be run has been assembled by Phase Three onto the SCRATCH AREA.

Line 3 contains a transfer command (TRA) to the address A + 2. Since the letter "A" always represents the line of code presently being processed (Line 3) this command transfers control to Line 5. It is executed next.

In Line 4 the WHAT "name" command, NAM, is used. This command stores the representation of the 5 characters of its operand into the next machine location. It essentially holds the word ALGOL ready for the operation to follow in Line 5.

Line 5 uses the "clear and add logic" machine instruction, CAL. Its operand is A-1, that is, the contents of the previous machine location. But the previous location contains the word ALGOL. Therefore, the word ALGOL is stored logically into the accumulator.

The code of Line 6 transfers control to the |23 routine. The purpose of bar-23 is to load and execute one of the language systems available to the user at Carnegie Tech. The single parameter to this routine is the name of the system located in the accumulator. Since ALGOL is located there, it is loaded and executed. Card images are drawn from the SCRATCH AREA, as previously specified by the bar-15 routine. Thus, the ALGOL program assembled on the AND SCRATCH AREA from HYDRO input code is executed.

APPENDIX II

SEGMENTS OF THE HYDRO COMPILER

```
      BEGIN
   SY SEGMENT 51
   SY RELEASE WHAT
   SY RELEASE SYMBOLIC LIBRARY
   AL COMMENT   THE HYDRO COMPILER - PHASE ONE, SECTION ONE;
           OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
           STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
           PAIRS,TEMP1,TEMP2;
           OWN LOGIC TYPEDATA;
           OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
           OWN LOGIC ARRAY IMAGE[1:81],IMAG[1:31],DATABUFFER[1:1620];
           OWN INTEGER ARRAY STACK[1:1000];
           OWN INTEGER ARRAY PROGRAM[1:500,1:2];
      PROCEDURE PHASE.ONE.RECOGNIZE;
           BEGIN
           LIBRARY PROCEDURE DISCWRITE,LINK;
           LABEL ERR1;
           PROCEDURE NEXT.COLUMN;
                BEGIN
                I ← I+1;
                     IF I > 80 THEN BEGIN
                     NAME(I→80(IMAGE[I]));
                     READ(<W>,→80<1A>);
                     IMAGE[81] ← '*';
                     STORED ← FALSE;
                     I ← 1;    END;
                END;
           PROCEDURE PR (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
                D11,D12,D13,D14,D15,D16,D17,D18,D19,D20,
                D21,D22,D23,D24,D25,D26,D27,D28,D29,D30);
                VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,
                D16,D17,D18,D19,D20,D21,D22,D23,D24,D25,D26,D27,D28,D29,D30;
                INTEGER N;
                LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16,
                D17.D18,D19,D20,D21,D22,D23,D24.D25,D26,D27,D28,D29,D30;
                BEGIN
                INTEGER LETTER;
                LOGIC PARAMETER;
                LETTER ← 1;
                FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
                     D11,D12,D13,D14,D15,D16,D17,D18,D19,D20,
                     D21,D22,D23,D24,D25,D26,D27,D28,D29,D30 DO
                     IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L100
                     ELSE LETTER ← LETTER+1;
```

```
        PARAMETER ← '*';
        L100:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
           IF STORED THEN GO TO L197;
           STACK[STNO] ← PROCBASE + N;
           IF STACK[STNO] = PROCBASE THEN GO TO L199;
           STNO ← STNO + 1;
           TRANSFER ← 1;
           SKIPOVER ← FALSE;
           I ← 80;
           GO TO L110;    END;
     END;
  PROCEDURE SR (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
     D11,D12,D13,D14,D15);
     VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
     INTEGER N;
     LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
     BEGIN
     INTEGER LETTER;
     LOGIC PARAMETER;
     LETTER ← 1;
     FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
        D11,D12,D13,D14,D15 DO
        IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L101
        ELSE LETTER ← LETTER + 1;
        PARAMETER ← '*';
        L101:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
           STACK[STNO] ← N;
           STNO ← STNO + 1;
           TRANSFER ← 1;
           SKIPOVER ← FALSE;
           TYPEDATA ← '    ';
           I ← K;
           GO TO L150;    END;
     END;
  PROCEDURE AR (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
     D11,D12,D13,D14,D15);
     VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
     INTEGER N;
     LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
     BEGIN
     INTEGER LETTER;
     LOGIC PARAMETER;
     LETTER ← 1;
     FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
        D11,D12,D13,D14,D15 DO
        IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L102
        ELSE LETTER ← LETTER+1;
        PARAMETER ← '*';
        L102:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
           STACK[STNO] ← ARRAYBASE + N;
           STNO ← STNO + 1;
           TRANSFER ← 1;
           SKIPOVER ← FALSE;
           TYPEDATA ← '    ';
           I ← K;
           GO TO L150;    END;
      END;
COMMENT   CODE FOR THE PHASE.ONE PROCEDURE;
   IF ARRAYBASE ≠ 10000 THEN BEGIN
      ARRAYBASE ← 10000;
      PROCBASE ← 20000;
      SPECBASE ← 30000;
      STORBASE ← 500;
      STNO ← STORBASE;
      DANO ← 1;
      DATABLOCK ← 1000;
      EXCESS ← 17;
      TEMP1 ← 1;
      TEMP2 ← 0;
      ERROR ← FALSE;
      I ← 80;
      TRANSFER ← 1;
      SKIPOVER ← FALSE;
      SEGS ← 2;
      END;
   L110: IF SKIPOVER THEN GO TO L115;
      L113: NEXT.COLUMN;
       IF IMAGE[I] = ' ' THEN GO TO L113;
      FOR J←I STEP 1 WHILE ¬(IMAGE[J] = ' ' ∨ IMAGE[J] = '←'
         ∨ IMAGE[J] = '[' ∨ J > 80) DO     ;
         IF J > 80 THEN GO TO L180;
      FOR K←I STEP 1 UNTIL J-1 DO
         IMAG[K-I+1] ← IMAGE[K];
         IMAG[K-I+1] ← '$';
      K ← J;
      L111:   FOR K←K STEP 1 WHILE ¬(IMAGE[K] = '←'
         ∨ IMAGE[K] = '[' ∨ K > 80) DO
         IF IMAGE[K] ≠ ' ' THEN GO TO L181;
      IF IMAGE[K] = '[' THEN BEGIN
         L112: K ← K+1;
         IF K=81 THEN GO TO L182;
         IF IMAGE[K] = ' ' ∨
            IMAGE[K] = '+' ∨
            IMAGE[K] = '-' ∨
            IMAGE[K] = '0' ∨
            IMAGE[K] = '1' ∨
            IMAGE[K] = '2' ∨
            IMAGE[K] = '3' ∨
            IMAGE[K] = '4' ∨
            IMAGE[K] = '5' ∨
            IMAGE[K] = '6' ∨
            IMAGE[K] = '7' ∨
            IMAGE[K] = '8' ∨
            IMAGE[K] = '9' ∨
            IMAGE[K] = ',' THEN GO TO L112;
         IF IMAGE[K] ≠ ']' THEN GO TO L183;
         K ← K+1;
         GO TO L111;
         END;
         IF K ≠ 81 THEN TYPE←1 ELSE TYPE←0;
      L115: IF TYPE = 1 THEN GO TO L130;
COMMENT CODE FOR PROCEDURE RECOGNITION;
      IF IMAGE[I] = 'A' THEN GO TO AP;
      IF IMAGE[I] = 'B' THEN GO TO BP;
      IF IMAGE[I] = 'C' THEN GO TO CP;
      IF IMAGE[I] = 'D' THEN GO TO DP;
      IF IMAGE[I] = 'E' THEN GO TO EP;
      IF IMAGE[I] = 'F' THEN GO TO FP;
      IF IMAGE[I] = 'G' THEN GO TO GP;
      IF IMAGE[I] = 'H' THEN GO TO HP;
      IF IMAGE[I] = 'I' THEN GO TO IP;
      IF IMAGE[I] = 'J' THEN GO TO JP;
      IF IMAGE[I] = 'K' THEN GO TO KP;
      IF IMAGE[I] = 'L' THEN GO TO LP;
      IF IMAGE[I] = 'M' THEN GO TO MP;
      IF IMAGE[I] = 'N' THEN GO TO NP;
      IF IMAGE[I] = 'O' THEN GO TO OP;
      IF IMAGE[I] = 'P' THEN GO TO PP;
      IF IMAGE[I] = 'Q' THEN GO TO QP;
      IF IMAGE[I] = 'R' THEN GO TO RP;
      IF IMAGE[I] = 'S' THEN GO TO SP;
      IF IMAGE[I] = 'T' THEN GO TO TP;
      IF IMAGE[I] = 'U' THEN GO TO UP;
      IF IMAGE[I] = 'V' THEN GO TO VP;
      IF IMAGE[I] = 'W' THEN GO TO WP;
      IF IMAGE[I] = 'X' THEN GO TO XP;
      IF IMAGE[I] = 'Y' THEN GO TO YP;
      IF IMAGE[I] = 'Z' THEN GO TO ZP;
      GO TO L184;
AP:   COMMENT   THE 'A' PROCEDURE CALLS;
PR( 1,'A','R','I','T','H','M','E','T','I','C','.','R','A','I','N',
      'F','A','L','L','.','A','V','E','R','A','G','E','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
BP:   COMMENT   THE 'B' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
CP:   COMMENT   THE 'C' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
DP:   COMMENT   THE 'D' PROCEDURE CALLS;
PR( 3,'D','A','T','A','.','O','R','D','E','R','I','N','G','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR( 5,'D','O','U','B','L','E','.','M','A','S','S','.','B','R','E',
      'A','K','.','C','H','E','C','K','*','*','*','*','*','*','*');
PR( 6,'D','U','R','A','T','I','O','N','.','C','U','R','V','E','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
EP:   COMMENT   THE 'E' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
FP:   COMMENT   THE 'F' PROCEDURE CALLS;
PR( 7,'F','R','E','Q','U','E','N','C','Y','.','C','U','R','V','E',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR( 8,'F','U','L','L','.','S','E','R','I','E','S','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
GP:   COMMENT   THE 'G' PROCEDURE CALLS;
PR( 9,'G','U','M','B','E','L','.','R','E','C','U','R','R','E','N',
        'C','E','.','I','N','T','E','R','V','A','L','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
HP:     COMMENT   THE 'H' PROCEDURE CALLS;
PR(10,'H','I','S','T','O','G','R','A','M','*','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(11,'H','I','S','T','O','G','R','A','M','.','D','U','R','A','T',
        'I','O','N','.','C','U','R','V','E','*','*','*','*','*','*');
PR(12,'H','I','S','T','O','G','R','A','M','.','F','R','E','Q','U',
        'E','N','C','Y','.','C','U','R','V','E','*','*','*','*','*');
PR(13,'H','Y','D','R','O','G','R','A','P','H','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(14,'H','Y','D','R','O','G','R','A','P','H','.','R','E','G','R',
        'E','S','S','I','O','N','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
IP:     COMMENT   THE 'I' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
JP:     COMMENT   THE 'J' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
KP:     COMMENT   THE 'K' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
LP:     COMMENT   THE 'L' PROCEDURE CALLS;
PR(15,'L','O','W','.','A','N','N','U','A','L','.','P','A','R','T',
        'I','A','L','.','S','E','R','I','E','S','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
MP:     COMMENT   THE 'M' PROCEDURE CALLS;
PR(16,'M','A','X','.','E','V','E','N','T','*','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(21,'M','E','A','N','.','E','V','E','N','T','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(22,'M','I','N','.','E','V','E','N','T','*','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(23,'M','I','S','S','I','N','G','.','R','A','I','N','.','C','H',
        'O','I','C','E','*','*','*','*','*','*','*','*','*','*','*');
PR(24,'M','I','S','S','I','N','G','.','R','A','I','N','.','N','O',
        'R','M','A','L','.','R','A','T','I','O','*','*','*','*','*');
PR(25,'M','I','S','S','I','N','G','.','R','A','I','N','.','S','T',
        'A','T','I','O','N','.','A','V','E','R','A','G','E','*','*');
PR(26,'M','U','L','T','I','P','L','E','.','R','E','G','R','E','S',
        'S','I','O','N','.','U','N','I','T','G','R','A','P','H','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
NP:     COMMENT   THE 'N' PROCEDURE CALLS;
PR(27,'N','O','R','M','A','L','.','A','N','N','U','A','L','.','P',
        'R','E','C','I','P','I','T','A','T','I','O','N','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
OP:     COMMENT   THE 'O' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
PP:     COMMENT   THE 'P' PROCEDURE CALLS;
PR( 0,'P','R','O','G','R','A','M','.','E','N','D','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(28,'P','A','R','T','I','A','L','.','S','E','R','I','E','S','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(29,'P','R','I','N','T','.','M','A','S','S','.','C','U','R','V',
        'E','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(30,'P','R','I','N','T','.','Y','E','A','R','L','Y','.','M','A',
        'S','S','.','C','U','R','V','E','*','*','*','*','*','*','*');
```

```
PR(82,'P','L','O','T','.','O','N','*','*','*','*','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(83,'P','L','O','T','.','O','F','F','*','*','*','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(84,'P','R','I','N','T','.','O','N','*','*','*','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(85,'P','R','I','N','T','.','O','F','F','*','*','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
QP:   COMMENT   THE 'Q' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
RP:   COMMENT   THE 'R' PROCEDURE CALLS;
PR(31,'R','A','I','N','F','A','L','L','.','D','O','U','B','L','E',
      '.','M','A','S','S','.','A','N','A','L','Y','S','I','S','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
SP:   COMMENT   THE 'S' PROCEDURE CALLS;
PR(32,'S','.','C','U','R','V','E','.','H','Y','D','R','O','G','R',
      'A','P','H','*','*','*','*','*','*','*','*','*','*','*','*');
PR(36,'S','T','A','N','D','A','R','D','.','D','E','V','I','A','T',
      'I','O','N','.','O','F','.','E','V','E','N','T','S','*','*');
PR(37,'S','T','A','N','D','A','R','D','.','R','E','C','U','R','R',
      'E','N','C','E','.','I','N','T','E','R','V','A','L','*','*');
PR(38,'S','T','D','.','U','N','I','T','G','R','A','P','H','.','M',
      'A','X','.','I','N','F','I','L','T','.','R','A','T','E','*');
PR(39,'S','T','D','.','U','N','I','T','G','R','A','P','H','.','V',
      'A','R','Y','.','I','N','F','I','L','T','.','R','A','T','E');
PR(40,'S','Y','N','T','H','E','T','I','C','.','U','N','I','T','G',
      'R','A','P','H','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
TP:   COMMENT   THE 'T' PROCEDURE CALLS;
PR(41,'T','H','I','E','S','S','E','N','.','R','A','I','N','F','A',
      'L','L','.','A','V','E','R','A','G','E','*','*','*','*','*');
PR(42,'T','I','M','E','.','S','E','R','I','E','S','*','*','*','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(43,'T','R','I','A','N','G','U','L','A','R','.','U','N','I','T',
      'G','R','A','P','H','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
UP:   COMMENT   THE 'U' PROCEDURE CALLS;
PR(44,'U','N','I','T','G','R','A','P','H','.','B','A','N','K','*',
      '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
VP:   COMMENT   THE 'V' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
WP:   COMMENT   THE 'W' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
XP:   COMMENT   THE 'X' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
YP:   COMMENT   THE 'Y' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
ZP:   COMMENT   THE 'Z' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
COMMENT   END OF CODE FOR PROCEDURE RECOGNITION;
COMMENT   CODE FOR DATA VARIABLE RECOGNITION;
L130: IF IMAGE[I] = 'A' THEN GO TO AD;
      IF IMAGE[I] = 'B' THEN GO TO BD;
```

```
        IF IMAGE[I] = 'C' THEN GO TO CD;
        IF IMAGE[I] = 'D' THEN GO TO DD;
        IF IMAGE[I] = 'E' THEN GO TO ED;
        IF IMAGE[I] = 'F' THEN GO TO FD;
        IF IMAGE[I] = 'G' THEN GO TO GD;
        IF IMAGE[I] = 'H' THEN GO TO HD;
        IF IMAGE[I] = 'I' THEN GO TO ID;
        IF IMAGE[I] = 'J' THEN GO TO JD;
        IF IMAGE[I] = 'K' THEN GO TO KD;
        IF IMAGE[I] = 'L' THEN GO TO LD;
        IF IMAGE[I] = 'M' THEN GO TO MD;
        IF IMAGE[I] = 'N' THEN GO TO ND;
        IF IMAGE[I] = 'O' THEN GO TO OD;
        IF IMAGE[I] = 'P' THEN GO TO PD;
        IF IMAGE[I] = 'Q' THEN GO TO QD;
        IF IMAGE[I] = 'R' THEN GO TO RD;
        IF IMAGE[I] = 'S' THEN GO TO SD;
        IF IMAGE[I] = 'T' THEN GO TO TD;
        IF IMAGE[I] = 'U' THEN GO TO UD;
        IF IMAGE[I] = 'V' THEN GO TO VD;
        IF IMAGE[I] = 'W' THEN GO TO WD;
        IF IMAGE[I] = 'X' THEN GO TO XD;
        IF IMAGE[I] = 'Y' THEN GO TO YD;
        IF IMAGE[I] = '7' THEN GO TO ZD;
        GO TO L186;
AD:     COMMENT   THE 'A' DATA VARIABLES;
SR( 1,'A','N','N','U','A','L','*','*','*','*','*','*','*','*','*');
SR( 2,'A','R','E','A','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
BD:     COMMENT   THE 'B' DATA VARIABLES;
SR( 3,'B','A','S','E','L','E','N','G','T','H','*','*','*','*','*');
AR( 1,'B','A','S','E','F','L','O','W','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
CD:     COMMENT   THE 'C' DATA VARIABLES;
SR( 4,'C','E','I','L','I','N','G','*','*','*','*','*','*','*','*');
SR( 6,'C','O','N','S','T','1','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
DD:     COMMENT   THE 'D' DATA VARIABLES;
SR( 8,'D','E','V','I','A','T','I','O','N','*','*','*','*','*','*');
SR( 9,'D','U','R','A','T','I','O','N','2','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ED:     COMMENT   THE 'E' DATA VARIABLES;
SR(10,'E','X','C','E','S','S','D','U','R','A','T','I','O','N','*');
AR( 4,'E','X','C','E','S','S','V','O','L','U','M','E','1','*','*');
AR( 5,'E','X','C','E','S','S','V','O','L','U','M','E','2','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
FD:     COMMENT   THE 'F' DATA VARIABLES;
AR( 3,'F','R','D','A','T','A','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
GD:     COMMENT   THE 'G' DATA VARIABLES;
SR(11,'G','R','I','D','A','R','E','A','*','*','*','*','*','*','*');
SR(12,'G','R','I','D','C','O','L','U','M','N','S','*','*','*','*');
SR(13,'G','R','I','D','R','O','W','S','*','*','*','*','*','*','*');
AR( 6,'G','R','A','P','H','A','R','R','A','Y','*','*','*','*','*');
AR( 7,'G','R','I','D','*','*','*','*','*','*','*','*','*','*','*');
```

```
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
HD:     COMMENT   THE 'H' DATA VARIABLES;
SR(14,'H','I','S','T','I','N','C','R','*','*','*','*','*','*','*');
AR( 8,'H','I','S','T','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ID:     COMMENT   THE 'I' DATA VARIABLES;
SR(15,'I','N','C','R','E','M','E','N','T','*','*','*','*','*','*');
SR(16,'I','N','I','T','I','A','L','1','*','*','*','*','*','*','*');
SR(17,'I','N','I','T','I','A','L','2','*','*','*','*','*','*','*');
SR(18,'I','N','I','T','I','A','L','3','*','*','*','*','*','*','*');
SR(19,'I','N','T','E','R','V','A','L','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
JD:     COMMENT   THE 'J' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
KD:     COMMENT   THE 'K' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
LD:     COMMENT   THE 'L' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
MD:     COMMENT   THE 'M' DATA VARIABLES;
SR(20,'M','A','X','M','I','N','*','*','*','*','*','*','*','*','*');
SR(21,'M','A','X','A','B','S','C','I','S','S','A','*','*','*','*');
SR(22,'M','E','A','N','V','A','L','U','E','*','*','*','*','*','*');
AR(12,'M','A','S','S','C','U','R','V','E','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ND:     COMMENT   THE 'N' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
OD:     COMMENT   THE 'O' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
PD:     COMMENT   THE 'P' DATA VARIABLES;
SR(23,'P','E','A','K','P','E','R','C','E','N','T','*','*','*','*');
SR(24,'P','E','R','C','E','N','T','L','E','N','G','T','H','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
QD:     COMMENT   THE 'Q' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
RD:     COMMENT   THE 'R' DATA VARIABLES;
AR(17,'R','A','I','N','*','*','*','*','*','*','*','*','*','*','*');
AR(18,'R','A','I','N','F','A','L','L','1','*','*','*','*','*','*');
AR(19,'R','A','I','N','F','A','L','L','2','*','*','*','*','*','*');
AR(20,'R','A','I','N','1','*','*','*','*','*','*','*','*','*','*');
AR(21,'R','A','I','N','2','*','*','*','*','*','*','*','*','*','*');
AR(22,'R','A','I','N','3','*','*','*','*','*','*','*','*','*','*');
AR(23,'R','U','N','O','F','F','1','*','*','*','*','*','*','*','*');
AR(24,'R','U','N','O','F','F','2','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
SD:     COMMENT   THE 'S' DATA VARIABLES;
SR(25,'S','H','A','P','E','F','A','C','T','O','R','*','*','*','*');
SR(26,'S','Y','N','A','R','E','A','*','*','*','*','*','*','*','*');
SR(27,'S','Y','N','I','N','C','R','E','M','E','N','T','*','*','*');
AR(25,'S','E','R','I','E','S','*','*','*','*','*','*','*','*','*');
AR(26,'S','H','E','D','R','A','I','N','*','*','*','*','*','*','*');
AR(27,'S','T','A','T','I','O','N','R','A','I','N','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
TD:     COMMENT   THE 'T' DATA VARIABLES;
SR( 5,'T','S','U','B','C','*','*','*','*','*','*','*','*','*','*');
SR(28,'T','I','M','E','U','N','I','T','*','*','*','*','*','*','*');
SR(29,'T','O','T','A','L','1','*','*','*','*','*','*','*','*','*');
SR(30,'T','O','T','A','L','2','*','*','*','*','*','*','*','*','*');
SR(31,'T','O','T','A','L','3','*','*','*','*','*','*','*','*','*');
SR(32,'T','O','T','A','L','4','*','*','*','*','*','*','*','*','*');
SR(33,'T','O','T','A','L','5','*','*','*','*','*','*','*','*','*');
SR(34,'T','O','T','A','L','6','*','*','*','*','*','*','*','*','*');
SR(35,'T','O','T','A','L','7','*','*','*','*','*','*','*','*','*');
SR(36,'T','O','T','A','L','8','*','*','*','*','*','*','*','*','*');
SR(63,'T','O','T','A','L','9','*','*','*','*','*','*','*','*','*');
SR(43,'T','O','T','A','L','2','0','*','*','*','*','*','*','*','*');
SR(44,'T','O','T','A','L','2','1','*','*','*','*','*','*','*','*');
SR(45,'T','O','T','A','L','2','2','*','*','*','*','*','*','*','*');
SR(46,'T','O','T','A','L','2','3','*','*','*','*','*','*','*','*');
SR(47,'T','O','T','A','L','2','4','*','*','*','*','*','*','*','*');
SR(48,'T','O','T','A','L','2','5','*','*','*','*','*','*','*','*');
SR(49,'T','O','T','A','L','2','6','*','*','*','*','*','*','*','*');
AR(30,'T','I','M','E','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
UD:    COMMENT   THE 'U' DATA VARIABLES;
AR(31,'U','N','I','T','G','R','A','P','H','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
VD:    COMMENT   THE 'V' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
WD:    COMMENT   THE 'W' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
XD:    COMMENT   THE 'X' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
YD:    COMMENT   THE 'Y' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ZD:    COMMENT   THE '7' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
COMMENT   END OF CODE FOR DATA VARIABLE RECOGNITION;
   L150:   NEXT.COLUMN;
       IF ¬ STORED THEN BEGIN
          NAME($FOR DANO←DANO STEP 1 UNTIL
          DANO+19 DO$(DATABUFFER[DANO]));
          READ(<1C>, ←20<4A>);
          DATABUFFER[DANO+1] ← '    ';
          DANO ← DANO+2;
          STORED ← TRUE;
          IF DANO ¬< 1600 THEN BEGIN
             DANO ← 1;
             DISCWRITE(1600,DATABUFFER[1],1,DATABLOCK,ERR1);
             DATABLOCK ← DATABLOCK+5;
             FOR DANO←DANO STEP 1 UNTIL EXCESS DO
             DATABUFFER[DANO] ← DATABUFFER[DANO+1600];
             EXCESS ← EXCESS-4;
             IF EXCESS = -4 THEN EXCESS ← 17;
             IF EXCESS = -3 THEN EXCESS ← 18;
             IF EXCESS = -2 THEN EXCESS ← 19;
             IF EXCESS = -1 THEN EXCESS ← 20;   END;
          END;
       IF IMAGE[I] = ' ' THEN GO TO L150;
          IF IMAGE[I] ≠ '+' ∧
             IMAGE[I] ≠ '-' ∧
```

```
         IMAGE[I] ≠ '0' ^
         IMAGE[I] ≠ '1' ^
         IMAGE[I] ≠ '2' ^
         IMAGE[I] ≠ '3' ^
         IMAGE[I] ≠ '4' ^
         IMAGE[I] ≠ '5' ^
         IMAGE[I] ≠ '6' ^
         IMAGE[I] ≠ '7' ^
         IMAGE[I] ≠ '8' ^
         IMAGE[I] ≠ '9' ^
         IMAGE[I] ≠ '.' ^
         IMAGE[I] ≠ '*' THEN GO TO L188;
      IF IMAGE[I] = '*' THEN GO TO L152;
   IF TYPEDATA = 'ALPH' THEN GO TO L189;
      TYPEDATA ← 'NUMR';
      IF IMAGE[I] = '+' ∨ IMAGE[I] = '-' THEN NEXT.COLUMN;
      HAVEPERIOD ← FALSE;
   L151: IF IMAGE[I] = '+' ∨ IMAGE[I] = '-' THEN GO TO L190;
      IF IMAGE[I] = '0' ∨
         IMAGE[I] = '1' ∨
         IMAGE[I] = '2' ∨
         IMAGE[I] = '3' ∨
         IMAGE[I] = '4' ∨
         IMAGE[I] = '5' ∨
         IMAGE[I] = '6' ∨
         IMAGE[I] = '7' ∨
         IMAGE[I] = '8' ∨
         IMAGE[I] = '9' THEN BEGIN
            IF I = 80 THEN GO TO L191;
            NEXT.COLUMN;
            GO TO L151;    END;
      IF IMAGE[I] = '.' THEN BEGIN
            IF I=80 THEN GO TO L191;
            IF HAVEPERIOD THEN GO TO L192;
            HAVEPERIOD ← TRUE;
            NEXT.COLUMN;
            GO TO L151;    END;
      FOR I←I STEP 1 WHILE IMAGE[I] = ' ' DO    ;
      IF IMAGE[I] = ';' THEN GO TO L110;
      IF IMAGE[I] = ',' ^ STACK[STNO-1] > ARRAYBASE
         THEN GO TO L150;
      GO TO L193;
   L152: IF TYPEDATA = 'NUMR' THEN GO TO L194;
         TYPEDATA ← 'ALPH';
         FOR I ← I+2 STEP 1 WHILE ¬(IMAGE[I] = '*') DO
         IF I > 79 THEN GO TO L195;
         I ← I+1;
      FOR I←I STEP 1 WHILE IMAGE[I] = ' ' DO    ;
      IF IMAGE[I] = ';' THEN GO TO  L110;
      IF IMAGE[I] = ',' ^ STACK[STNO-1] > ARRAYBASE
         THEN GO TO L150;
      GO TO L196;
L160: TRANSFER ← TRANSFER+1;
      SKIPOVER ← TRUE;
      LINK(52);
```

```
COMMENT   PHASE ONE ERROR MESSAGES;
L180: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'A PROCEDURE CALL OR DATA VARIABLE WAS NOT COMPLETED ',E,
      'WITHIN THE BOUNDS OF A SINGLE CARD',E>);
      GO TO L198;
L181: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'THERE ARE BLANK SPACES WITHIN A PROCEDURE CALL OR ',
      'DATA VARIABLE',E,'OR ELSE THERE IS SOMETHING ON THE CARD ',
      'AFTER A LEGAL PROCEDURE CALL',E,'OR ELSE A LEFT ARROW IS ',
      'MISSING ON A DATA CARD',E>);
      GO TO L198;
L182: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'SUBSCRIPT BRACKETS ARE NOT MATCHED ON A ',
      'SINGLE DATA CARD',E>);
      GO TO L198;
L183: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'AN INTEGER, COMMA, OR RIGHT BRACKET WAS EXPECTED',E,
      'BUT SOMETHING ELSE WAS FOUND INSTEAD',E>);
      GO TO L198;
L184: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'THE FIRST CHARACTER OF THE ABOVE PROCEDURE CALL IS NOT ',
      'A LEGAL LETTER',E,'OR ELSE SOMETHING BESIDES A LEGAL ',
      'PROCEDURE CALL OR VARIABLE',E,'FOLLOWS A SEMICOLON',E>);
      GO TO L198;
L185: PRINT(<120B,1C,'THE ABOVE CARD HAS AN IMPROPER PROCEDURE ',
      'CALL OR DATA VARIABLE',E,'THE PROCEDURE CALL IS EITHER ',
      'SPELLED INCORRECTLY, OR A PERIOD IS MISSING',E,'OR IT HAS',
      ' NOT BEEN COMPLETED ON A SINGLE CARD',E,'THE DATA ',
      'VARIABLE MAY NOT HAVE BEEN FOLLOWED BY A LEFT ARROW',E,
      'ON THE SAME CARD AS ITSELF',E>);
      GO TO L198;
L186: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'THE FIRST CHARACTER OF THE ABOVE DATA VARIABLE IS NOT ',
      'A LEGAL LETTER',E>);
      GO TO L198;
L187: PRINT(<120B,1C,'THE ABOVE CARD HAS AN IMPROPER DATA ',
      'VARIABLE',E,'IT IS PROBABLY SPELLED INCORRECTLY',E>);
      GO TO L198;
L188: PRINT(<120B,1C,'THERE IS AN ERROR ON ONE OF THE ABOVE ',
      'TWO CARDS',E,'AN ILLEGAL PIECE OF DATA IS ',
      'LOCATED ON THE ABOVE CARD',E,'OR ELSE A SEMICOLON ',
      'SHOULD HAVE BEEN PLACED',E,'AT THE END OF DATA ',
      'ON ONE OF THE ABOVE TWO CARDS',E>);
      GO TO L198;
L189: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'BOTH ALPHANUMERIC AND NUMERIC DATA ',
      'ARE GIVEN TO THE SAME ARRAY',E>);
      GO TO L198;
L190: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'A NUMBER HAS MORE THAN ONE PLUS OR MINUS SIGN',E,
      'OR ELSE THE SIGN IS NOT IN FRONT OF THE NUMBER',E>);
      GO TO L198;
L191: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
      'A DATA CARD SHOULD END WITH EITHER A COMMA OR ',
      'A SEMICOLON',E>);
```

Reproduced from
best available copy.

```
       GO TO L198;
   L192: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'A PIECE OF NUMERICAL DATA HAS MORE THAN ONE ',
       'DECIMAL POINT',E>);
       GO TO L198;
   L193: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'A PIECE OF NUMERICAL DATA WAS NOT FOLLOWED BY A ',
       'COMMA OR A SEMICOLON',E,'OR ELSE AN ILLEGAL PIECE ',
       'OF DATA WAS ENTERED',E,'OR ELSE MORE THAN ONE PIECE OF ',
       'DATA',E,'WAS GIVEN TO A NUMERICAL SINGLE VARIABLE',E>);
       GO TO L198;
   L194: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'BOTH NUMERIC AND ALPHANUMERIC DATA ',
       'ARE GIVEN TO THE SAME ARRAY',E>);
       GO TO L198;
   L195: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'THE END OF THE CARD WAS REACHED',E,
       'BEFORE A PIECE OF ALPHANUMERIC DATA WAS COMPLETED',E>);
       GO TO L198;
   L196: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'A PIECE OF ALPHANUMERICAL DATA WAS NOT FOLLOWED BY A ',
       'COMMA OR SEMICOLON',E,'OR ELSE MORE THAN ONE PIECE OF ',
       'DATA',E,'WAS GIVEN TO AN ALPHANUMERIC SINGLE VARIABLE',E>);
       GO TO L198;
   L197: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
       'A PROCEDURE CALL HAS BEEN PLACED ON A DATA CARD',E,
       'PROCEDURE CALLS MUST OCCUPY CARDS BY THEMSELVES',E>);
   L198: ERROR ← TRUE;
       TRANSFER ← 1;
       SKIPOVER ← FALSE;
       I ← 80;
       GO TO L110;
   L199: DATABUFFER[DANO] ← '←←←←';
   DISCWRITE(1600,DATABUFFER[1],1,DATABLOCK,ERR1);
   DATABLOCK ← DATABLOCK+5;
   IF ¬ ERROR THEN BEGIN
      OPNO ← STORBASE;
      PAIRS ← 0;
      DATAHEAD ← FALSE;
      LINK (53);
      END ELSE HALT;
   ERR1: PRINT(<'PHASE ONE COMPILER ERROR',E>);
   END;
PHASE.ONE.RECOGNIZE;
END;
BEGIN
PRINT(<'THE SEGMENT IS STORED',E>);
END;




BEGIN
SY SEGMENT 52
SY RELEASE WHAT
SY RELEASE SYMBOLIC LIBRARY
AL COMMENT   THE HYDRO COMPILER - PHASE ONE, SECTION TWO;
      OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
      STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
      PAIRS,TEMP1,TEMP2;
      OWN LOGIC TYPEDATA;
      OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
      OWN LOGIC ARRAY IMAGE[1:81],IMAG[1:31],DATABUFFER[1:1620];
      OWN INTEGER ARRAY STACK[1:1000];
      OWN INTEGER ARRAY PROGRAM[1:500,1:2];
   PROCEDURE PHASE.ONE.RECOGNIZE;
      BEGIN
      LIBRARY PROCEDURE DISCWRITE,LINK;
      LABEL ERR1;
      PROCEDURE NEXT.COLUMN;
         BEGIN
         I ← I+1;
            IF I > 80 THEN BEGIN
            NAME(I→80(IMAGE[I]));
            READ(<W>,→80<1A>);
            IMAGE[81] ← '*';
            STORED ← FALSE;
            I ← 1;    END;
         END;
      PROCEDURE PR (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
         D11,D12,D13,D14,D15,D16,D17,D18,D19,D20,
         D21,D22,D23,D24,D25,D26,D27,D28,D29,D30);
         VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,
         D16,D17,D18,D19,D20,D21,D22,D23,D24,D25,D26,D27,D28,D29,D30;
         INTEGER N;
         LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16,
         D17,D18,D19,D20,D21,D22,D23,D24,D25,D26,D27,D28,D29,D30;
         BEGIN
         INTEGER LETTER;
         LOGIC PARAMETER;
         LETTER ← 1;
         FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
            D11,D12,D13,D14,D15,D16,D17,D18,D19,D20,
            D21,D22,D23,D24,D25,D26,D27,D28,D29,D30 DO
            IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L100
            ELSE LETTER ← LETTER+1;
            PARAMETER ← '*';
            L100:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
               IF STORED THEN GO TO L197;
               STACK[STNO] ← PROCBASE + N;
               STNO ← STNO + 1;
               TRANSFER ← 1;
               SKIPOVER ← FALSE;
               I ← 80;
               GO TO L110;    END;
         END;
      PROCEDURE SR (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
         D11,D12,D13,D14,D15);
         VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
      INTEGER N;
      LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
      BEGIN
      INTEGER LETTER;
      LOGIC PARAMETER;
      LETTER ← 1;
      FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
         D11,D12,D13,D14,D15 DO
         IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L101
         ELSE LETTER ← LETTER + 1;
         PARAMETER ← '*';
         L101:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
            STACK[STNO] ← N;
            STNO ← STNO + 1;
            TRANSFER ← 1;
            SKIPOVER ← FALSE;
            TYPEDATA ← '    ';
            I ← K;
            GO TO L150;   END;
      END;
   PROCEDURE AH (N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
      D11,D12,D13,D14,D15);
      VALUE N,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
      INTEGER N;
      LOGIC D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15;
      BEGIN
      INTEGER LETTER;
      LOGIC PARAMETER;
      LETTER ← 1;
      FOR PARAMETER ← D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,
         D11,D12,D13,D14,D15 DO
         IF PARAMETER ≠ IMAG[LETTER] THEN GO TO L102
         ELSE LETTER ← LETTER+1;
         PARAMETER ← '*';
         L102:  IF IMAG[LETTER] = '$' ∧ PARAMETER = '*' THEN BEGIN
            STACK[STNO] ← ARRAYBASE + N;
            STNO ← STNO + 1;
            TRANSFER ← 1;
            SKIPOVER ← FALSE;
            TYPEDATA ← '    ';
            I ← K;
            GO TO L150;   END;
      END;
COMMENT   CODE FOR THE PHASE.ONE PROCEDURE;
   L110: IF SKIPOVER THEN GO TO L115;
      L113: NEXT.COLUMN;
         IF IMAGE[I] = ' ' THEN GO TO L113;
      FOR J←I STEP 1 WHILE ¬(IMAGE[J] = ' ' ∨ IMAGE[J] = '←'
         ∨ IMAGE[J] = '[' ∨ J > 80) DO    ;
         IF J > 80 THEN GO TO L180;
      FOR K←I STEP 1 UNTIL J-1 DO
         IMAG[K-I+1] ← IMAGE[K];
         IMAG[K-I+1] ← '$';
      K ← J;
      L111:   FOR K←K STEP 1 WHILE ¬(IMAGE[K] = '←'
      ∨ IMAGE(K) = '(' ∨ K > 80) DO
      IF IMAGE(K) ≠ ' ' THEN GO TO L181;
    IF IMAGE(K) = '(' THEN BEGIN
      L112: K ← K+1;
      IF K=81 THEN GO TO L182;
      IF IMAGE(K) = ' ' ∨
        IMAGE(K) = '+' ∨
        IMAGE(K) = '-' ∨
        IMAGE(K) = '0' ∨
        IMAGE(K) = '1' ∨
        IMAGE(K) = '2' ∨
        IMAGE(K) = '3' ∨
        IMAGE(K) = '4' ∨
        IMAGE(K) = '5' ∨
        IMAGE(K) = '6' ∨
        IMAGE(K) = '7' ∨
        IMAGE(K) = '8' ∨
        IMAGE(K) = '9' ∨
        IMAGE(K) = ',' THEN GO TO L112;
      IF IMAGE(K) ≠ ')' THEN GO TO L183;
      K ← K+1;
      GO TO L111;
      END;
      IF K ≠ 81 THEN TYPE←1 ELSE TYPE←0;
   L115: IF TYPE = 1 THEN GO TO L130;
COMMENT CODE FOR PROCEDURE RECOGNITION;
    IF IMAGE(I) = 'A' THEN GO TO AP;
    IF IMAGE(I) = 'B' THEN GO TO BP;
    IF IMAGE(I) = 'C' THEN GO TO CP;
    IF IMAGE(I) = 'D' THEN GO TO DP;
    IF IMAGE(I) = 'E' THEN GO TO EP;
    IF IMAGE(I) = 'F' THEN GO TO FP;
    IF IMAGE(I) = 'G' THEN GO TO GP;
    IF IMAGE(I) = 'H' THEN GO TO HP;
    IF IMAGE(I) = 'I' THEN GO TO IP;
    IF IMAGE(I) = 'J' THEN GO TO JP;
    IF IMAGE(I) = 'K' THEN GO TO KP;
    IF IMAGE(I) = 'L' THEN GO TO LP;
    IF IMAGE(I) = 'M' THEN GO TO MP;
    IF IMAGE(I) = 'N' THEN GO TO NP;
    IF IMAGE(I) = 'O' THEN GO TO OP;
    IF IMAGE(I) = 'P' THEN GO TO PP;
    IF IMAGE(I) = 'Q' THEN GO TO QP;
    IF IMAGE(I) = 'R' THEN GO TO RP;
    IF IMAGE(I) = 'S' THEN GO TO SP;
    IF IMAGE(I) = 'T' THEN GO TO TP;
    IF IMAGE(I) = 'U' THEN GO TO UP;
    IF IMAGE(I) = 'V' THEN GO TO VP;
    IF IMAGE(I) = 'W' THEN GO TO WP;
    IF IMAGE(I) = 'X' THEN GO TO XP;
    IF IMAGE(I) = 'Y' THEN GO TO YP;
    IF IMAGE(I) = 'Z' THEN GO TO ZP;
    GO TO L184;
AP: COMMENT  THE 'A' PROCEDURE CALLS;
PR(56,'A','R','E','A','.','A','L','P','H','A','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(57,'A','R','E','A','.','B','E','T','A','*','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(58,'A','R','E','A','.','C','E','N','D','I','S','T','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(59,'A','R','E','A','.','H','Y','D','R','A','D','.','A','V','G',
        'M','A','N','N','*','*','*','*','*','*','*','*','*','*','*');
PR(53,'A','V','E','R','A','G','E','.','M','A','N','N','I','N','G',
        '.','N','*','*','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
BP:     COMMENT   THE 'B' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
CP:     COMMENT   THE 'C' PROCEDURE CALLS;
PR(52,'C','E','N','T','R','O','I','D','A','L','.','D','I','S','T',
        'A','N','C','E','*','*','*','*','*','*','*','*','*','*','*');
PR(66,'C','H','A','N','N','E','L','.','R','O','U','G','H','N','E',
        'S','S','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(50,'C','O','N','V','E','Y','A','N','C','E','S','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(72,'C','R','I','T','I','C','A','L','.','D','E','P','T','H','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(47,'C','R','O','S','S','.','S','E','C','T','I','O','N','.','A',
        'R','E','A','*','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
DP:     COMMENT   THE 'D' PROCEDURE CALLS;
PR(63,'D','I','S','C','H','A','R','G','E','*','*','*','*','*','*',
        '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR( 4,'D','E','P','T','H','.','A','R','E','A','.','C','U','R','V',
        'E','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(17,'D','E','P','T','H','.','A','R','E','A','.','D','U','R','A',
        'T','I','O','N','.','C','U','R','V','E','S','*','*','*','*');
PR(18,'D','E','P','T','H','.','D','U','R','A','T','I','O','N','.',
        'C','O','M','B','I','N','E','D','.','Z','O','N','E','S','*');
PR(19,'D','E','P','T','H','.','D','U','R','A','T','I','O','N','.',
        'S','I','N','G','L','E','.','Z','O','N','E','S','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
EP:     COMMENT   THE 'E' PROCEDURE CALLS;
PR(49,'E','N','E','R','G','Y','.','C','O','E','F','F','I','C','I',
        'E','N','T','*','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
FP:     COMMENT   THE 'F' PROCEDURE CALLS;
PR(74,'F','L','O','W','.','P','R','O','F','I','L','E','S','.','C',
        'O','N','T','R','O','L','.','P','T','*','*','*','*','*','*');
PR(69,'F','R','I','C','T','I','O','N','.','H','E','A','D','.','L',
        'O','S','S','*','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
GP:     COMMENT   THE 'G' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
HP:     COMMENT   THE 'H' PROCEDURE CALLS;
PR(51,'H','Y','D','R','A','U','L','I','C','.','R','A','D','I','U',
        'S','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
IP:     COMMENT   THE 'I' PROCEDURE CALLS;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
JP:     COMMENT   THE 'J' PROCEDURE CALLS;
```

```
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
KP:    COMMENT   THE 'K' PROCEDURE CALLS;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
LP:    COMMENT   THE 'L' PROCEDURE CALLS;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
MP:    COMMENT   THE 'M' PROCEDURE CALLS;
PR(62,'M','E','A','N','.','V','E','L','O','C','I','T','Y','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(70,'M','O','M','E','N','T','U','M','*','*','*','*','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(54,'M','O','M','E','N','T','U','M','.','C','O','E','F','F','I',
       'C','I','E','N','T','*','*','*','*','*','*','*','*','*','*');
PR(20,'M','A','X','I','M','U','M','.','S','T','A','T','I','O','N',
       '.','P','R','E','C','I','P','I','T','A','T','I','O','N','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
NP:    COMMENT   THE 'N' PROCEDURE CALLS;
PR(71,'N','O','R','M','A','L','.','D','E','P','T','H','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(64,'N','O','R','M','A','L','.','D','I','S','C','H','A','R','G',
       'E','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
OP:    COMMENT   THE 'O' PROCEDURE CALLS;
PR(75,'O','P','E','N','.','C','H','A','N','N','E','L','.','P','L',
       'O','W','.','P','R','O','F','I','L','E','S','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
PP:    COMMENT   THE 'P' PROCEDURE CALLS;
PR(68,'P','R','E','S','S','U','R','E','.','F','O','R','C','E','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
QP:    COMMENT   THE 'Q' PROCEDURE CALLS;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
RP:    COMMENT   THE 'R' PROCEDURE CALLS;
PR(60,'R','E','A','C','H','.','L','E','N','G','T','H','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(61,'R','E','A','C','H','.','S','L','O','P','E','*','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
SP:    COMMENT   THE 'S' PROCEDURE CALLS;
PR(55,'S','E','C','T','I','O','N','.','E','L','E','M','E','N','T',
       'S','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(73,'S','E','Q','U','E','N','T','.','D','E','P','T','H','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(65,'S','L','U','P','E','S','*','*','*','*','*','*','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(67,'S','P','E','C','I','F','I','C','.','E','N','E','R','G','Y',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
PR(33,'S','I','X','.','H','O','U','R','.','M','A','S','S','C','U',
       'R','V','E','S','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
TP:    COMMENT   THE 'T' PROCEDURE CALLS;
PR(46,'T','O','P','W','I','D','T','H','S','*','*','*','*','*','*',
       '*','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
UP:    COMMENT   THE 'U' PROCEDURE CALLS;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
```

```
VP:   COMMENT   THE 'V' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
WP:   COMMENT   THE 'W' PROCEDURE CALLS;
PR(48,'W','E','T','T','E','D','.','P','E','R','I','M','E','T','E',
      'R','*','*','*','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
XP:   COMMENT   THE 'X' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
YP:   COMMENT   THE 'Y' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
ZP:   COMMENT   THE 'Z' PROCEDURE CALLS;
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L185;
COMMENT   END OF CODE FOR PROCEDURE RECOGNITION;
COMMENT   CODE FOR DATA VARIABLE RECOGNITION;
L130: IF IMAGE[I] = 'A' THEN GO TO AD;
      IF IMAGE[I] = 'B' THEN GO TO BD;
      IF IMAGE[I] = 'C' THEN GO TO CD;
      IF IMAGE[I] = 'D' THEN GO TO DD;
      IF IMAGE[I] = 'E' THEN GO TO ED;
      IF IMAGE[I] = 'F' THEN GO TO FD;
      IF IMAGE[I] = 'G' THEN GO TO GD;
      IF IMAGE[I] = 'H' THEN GO TO HD;
      IF IMAGE[I] = 'I' THEN GO TO ID;
      IF IMAGE[I] = 'J' THEN GO TO JD;
      IF IMAGE[I] = 'K' THEN GO TO KD;
      IF IMAGE[I] = 'L' THEN GO TO LD;
      IF IMAGE[I] = 'M' THEN GO TO MD;
      IF IMAGE[I] = 'N' THEN GO TO ND;
      IF IMAGE[I] = 'O' THEN GO TO OD;
      IF IMAGE[I] = 'P' THEN GO TO PD;
      IF IMAGE[I] = 'Q' THEN GO TO QD;
      IF IMAGE[I] = 'R' THEN GO TO RD;
      IF IMAGE[I] = 'S' THEN GO TO SD;
      IF IMAGE[I] = 'T' THEN GO TO TD;
      IF IMAGE[I] = 'U' THEN GO TO UD;
      IF IMAGE[I] = 'V' THEN GO TO VD;
      IF IMAGE[I] = 'W' THEN GO TO WD;
      IF IMAGE[I] = 'X' THEN GO TO XD;
      IF IMAGE[I] = 'Y' THEN GO TO YD;
      IF IMAGE[I] = 'Z' THEN GO TO ZD;
      GO TO L186;
AD:   COMMENT   THE 'A' DATA VARIABLES;
AR(35,'A','L','P','H','A','*','*','*','*','*','*','*','*','*','*');
AR(36,'A','R','T','M','A','T','R','I','X','*','*','*','*','*','*');
AR(37,'A','V','G','M','A','N','N','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
BD:   COMMENT   THE 'B' DATA VARIABLES;
SR(50,'B','E','G','S','T','A','T','*','*','*','*','*','*','*','*');
AR(38,'B','E','T','A','*','*','*','*','*','*','*','*','*','*','*');
      IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
CD:   COMMENT   THE 'C' DATA VARIABLES;
AR(39,'C','E','N','D','I','S','T','*','*','*','*','*','*','*','*');
AR(40,'C','R','O','S','S','A','R','E','A','*','*','*','*','*','*');
SR( 7,'C','O','N','V','E','R','T','E','R','*','*','*','*','*','*');
AR( 2,'C','E','N','T','E','R','2','*','*','*','*','*','*','*','*');
```

```
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
DD:     COMMENT   THE 'D' DATA VARIABLES;
SR(51,'D','E','P','T','H','1','*','*','*','*','*','*','*','*','*');
AR(55,'D','E','P','T','H','2','*','*','*','*','*','*','*','*','*');
AR(41,'D','I','S','C','H','A','R','G','E','1','*','*','*','*','*');
AR(42,'D','I','S','T','A','N','C','E','1','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ED:     COMMENT   THE 'E' DATA VARIABLES;
SR(52,'E','L','E','V','A','T','I','O','N','1','*','*','*','*','*');
SR(53,'E','N','D','S','T','A','T','*','*','*','*','*','*','*','*');
AR(43,'E','L','E','V','A','T','I','O','N','2','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
FD:     COMMENT   THE 'F' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
GD:     COMMENT   THE 'G' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
HD:     COMMENT   THE 'H' DATA VARIABLES;
AR(44,'H','Y','D','R','A','D','*','*','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ID:     COMMENT   THE 'I' DATA VARIABLES;
AR(56,'I','N','P','U','T','D','E','P','T','H','*','*','*','*','*');
AR( 9,'I','S','O','H','Y','E','T','2','*','*','*','*','*','*','*');
AR(10,'I','S','O','H','Y','E','T','3','*','*','*','*','*','*','*');
AR(11,'I','S','O','H','Y','E','T','4','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
JD:     COMMENT   THE 'J' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
KD:     COMMENT   THE 'K' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
LD:     COMMENT   THE 'L' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
MD:     COMMENT   THE 'M' DATA VARIABLES;
SR(54,'M','A','X','A','R','E','A','S','*','*','*','*','*','*','*');
SR(55,'M','A','X','P','O','I','N','T','S','*','*','*','*','*','*');
SR(56,'M','E','S','S','A','G','E','1','*','*','*','*','*','*','*');
SR(57,'M','E','S','S','A','G','E','2','*','*','*','*','*','*','*');
SR(58,'M','E','S','S','A','G','E','3','*','*','*','*','*','*','*');
SR(59,'M','E','S','S','A','G','E','7','*','*','*','*','*','*','*');
AR(45,'M','A','N','N','*','*','*','*','*','*','*','*','*','*','*');
AR(13,'M','A','S','S','I','N','P','U','T','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ND:     COMMENT   THE 'N' DATA VARIABLES;
SR(60,'N','O','A','R','T','*','*','*','*','*','*','*','*','*','*');
SR(61,'N','O','D','I','S','C','H','*','*','*','*','*','*','*','*');
AR(46,'N','O','A','R','E','A','S','*','*','*','*','*','*','*','*');
AR(47,'N','O','P','O','I','N','T','S','*','*','*','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
OD:     COMMENT   THE 'O' DATA VARIABLES;
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
PD:     COMMENT   THE 'P' DATA VARIABLES;
AR(14,'P','L','A','N','I','M','E','T','E','R','2','*','*','*','*');
AR(15,'P','L','A','N','I','M','E','T','E','R','3','*','*','*','*');
AR(16,'P','L','A','N','I','M','E','T','E','R','4','*','*','*','*');
        IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
QD:     COMMENT   THE 'Q' DATA VARIABLES;
```

```
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
RD:    COMMENT   THE 'R' DATA VARIABLES;
AR(48,'R','C','H','L','E','N','G','T','H','*','*','*','*','*','*');
AR(49,'R','C','H','S','L','O','P','E','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
SD:    COMMENT   THE 'S' DATA VARIABLES;
SR(62,'S','P','E','C','W','T','*','*','*','*','*','*','*','*','*');
AR(50,'S','T','A','T','C','O','D','E','*','*','*','*','*','*','*');
AR(28,'S','T','A','T','I','O','N','1','*','*','*','*','*','*','*');
AR(29,'S','T','A','T','I','O','N','3','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
TD:    COMMENT   THE 'T' DATA VARIABLES;
SR(37,'T','O','T','A','L','1','0','*','*','*','*','*','*','*','*');
SR(38,'T','O','T','A','L','1','1','*','*','*','*','*','*','*','*');
SR(39,'T','O','T','A','L','1','2','*','*','*','*','*','*','*','*');
SR(40,'T','O','T','A','L','1','3','*','*','*','*','*','*','*','*');
SR(41,'T','O','T','A','L','1','4','*','*','*','*','*','*','*','*');
SR(42,'T','O','T','A','L','1','5','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
UD:    COMMENT   THE 'U' DATA VARIABLES;
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
VD:    COMMENT   THE 'V' DATA VARIABLES;
AR(51,'V','E','L','O','C','I','T','Y','1','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
WD:    COMMENT   THE 'W' DATA VARIABLES;
AR(52,'W','S','E','L','*','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
XD:    COMMENT   THE 'X' DATA VARIABLES;
AR(53,'X','V','A','L','U','E','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
YD:    COMMENT   THE 'Y' DATA VARIABLES;
AR(54,'Y','V','A','L','U','E','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
ZD:    COMMENT   THE 'Z' DATA VARIABLES;
AR(32,'Z','O','N','E','1','*','*','*','*','*','*','*','*','*','*');
AR(33,'Z','O','N','E','3','*','*','*','*','*','*','*','*','*','*');
AR(34,'Z','O','N','E','4','*','*','*','*','*','*','*','*','*','*');
       IF TRANSFER < SEGS THEN GO TO L160 ELSE GO TO L187;
COMMENT   END OF CODE FOR DATA VARIABLE RECOGNITION;
   L150:   NEXT.COLUMN;
       IF ¬ STORED THEN BEGIN
          NAME(<FOR DANO←DANO STEP 1 UNTIL
          DANO+19 DO>(DATABUFFER[DANO]));
          READ(<1C>, ←20<44>);
          DATABUFFER[DANO+1] ← '      ';
          DANO ← DANO+2;
          STORED ← TRUE;
          IF DANO ¬< 1600 THEN BEGIN
             DANO ← 1;
             DISCWRITE(1600,DATABUFFER[1],1,DATABLOCK,ERR1);
             DATABLOCK ← DATABLOCK+5;
             FOR DANO←DANO STEP 1 UNTIL EXCESS DO
             DATABUFFER[DANO] ← DATABUFFER[DANO+1600];
             EXCESS ← EXCESS-4;
             IF EXCESS = -4 THEN EXCESS ← 17;
```

```
    IF EXCESS = -3 THEN EXCESS ← 18;
    IF EXCESS = -2 THEN EXCESS ← 19;
    IF EXCESS = -1 THEN EXCESS ← 20;    END;
  END;
IF IMAGE[I] = ' ' THEN GO TO L150;
  IF IMAGE[I] ≠ '+' ∧
    IMAGE[I] ≠ '-' ∧
    IMAGE[I] ≠ '0' ∧
    IMAGE[I] ≠ '1' ∧
    IMAGE[I] ≠ '2' ∧
    IMAGE[I] ≠ '3' ∧
    IMAGE[I] ≠ '4' ∧
    IMAGE[I] ≠ '5' ∧
    IMAGE[I] ≠ '6' ∧
    IMAGE[I] ≠ '7' ∧
    IMAGE[I] ≠ '8' ∧
    IMAGE[I] ≠ '9' ∧
    IMAGE[I] ≠ '.' ∧
    IMAGE[I] ≠ '*' THEN GO TO L188;
  IF IMAGE[I] = '*' THEN GO TO L152;
IF TYPEDATA = 'ALPH' THEN GO TO L189;
  TYPEDATA ← 'NUMR';
  IF IMAGE[I] = '+' ∨ IMAGE[I] = '-' THEN NEXT.COLUMN;
  HAVEPERIOD ← FALSE;
L151: IF IMAGE[I] = '+' ∨ IMAGE[I] = '-' THEN GO TO L190;
  IF IMAGE[I] = '0' ∨
    IMAGE[I] = '1' ∨
    IMAGE[I] = '2' ∨
    IMAGE[I] = '3' ∨
    IMAGE[I] = '4' ∨
    IMAGE[I] = '5' ∨
    IMAGE[I] = '6' ∨
    IMAGE[I] = '7' ∨
    IMAGE[I] = '8' ∨
    IMAGE[I] = '9' THEN BEGIN
      IF I = 80 THEN GO TO L191;
      NEXT.COLUMN;
      GO TO L151;    END;
  IF IMAGE[I] = '.' THEN BEGIN
      IF I=80 THEN GO TO L191;
      IF HAVEPERIOD THEN GO TO L192;
      HAVEPERIOD ← TRUE;
      NEXT.COLUMN;
      GO TO L151;    END;
  FOR I←I STEP 1 WHILE IMAGE[I] = ' ' DO    ;
  IF IMAGE[I] = ';' THEN GO TO L110;
  IF IMAGE[I] = ',' ∧ STACK[STNO-1] > ARRAYBASE
    THEN GO TO L150;
  GO TO L193;
L152: IF TYPEDATA = 'NUMR' THEN GO TO L194;
    TYPEDATA ← 'ALPH';
    FOR I ← I+2 STEP 1 WHILE ¬(IMAGE[I] = '*') DO
    IF I > 79 THEN GO TO L195;
    I ← I+1;
  FOR I←I STEP 1 WHILE IMAGE[I] = ' ' DO    ;
```

```
        IF IMAGE[1] = ';' THEN GO TO  L110;
        IF IMAGE[1] = ',' ^ STACK[STNO-1] > ARRAYBASE
           THEN GO TO L150;
        GO TO L198;
L160:   TRANSFER ← TRANSFER+1;
        SKIPOVER ← TRUE:
        LINK(51);
COMMENT   PHASE ONE ERROR MESSAGES;
L180:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'A PROCEDURE CALL OR DATA VARIABLE WAS NOT COMPLETED ',E,
        'WITHIN THE BOUNDS OF A SINGLE CARD',E>);
        GO TO L198;
L181:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'THERE ARE BLANK SPACES WITHIN A PROCEDURE CALL OR ',
        'DATA VARIABLE',E,'OR ELSE THERE IS SOMETHING ON THE CARD ',
        'AFTER A LEGAL PROCEDURE CALL',E,'OR ELSE A LEFT ARROW IS ',
        'MISSING ON A DATA CARD',E>);
        GO TO L198;
L182:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'SUBSCRIPT BRACKETS ARE NOT MATCHED ON A ',
        'SINGLE DATA CARD',E>);
        GO TO L198;
L183:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'AN INTEGER, COMMA, OR RIGHT BRACKET WAS EXPECTED',E,
        'BUT SOMETHING ELSE WAS FOUND INSTEAD',E>);
        GO TO L198;
L184:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'THE FIRST CHARACTER OF THE ABOVE PROCEDURE CALL IS NOT ',
        'A LEGAL LETTER',E,'OR ELSE SOMETHING BESIDES A LEGAL ',
        'PROCEDURE CALL OR VARIABLE',E,'FOLLOWS A SEMICOLON',E>);
        GO TO L198;
L185:   PRINT(<120B,1C,'THE ABOVE CARD HAS AN IMPROPER PROCEDURE ',
        'CALL OR DATA VARIABLE',E,'THE PROCEDURE CALL IS EITHER ',
        'SPELLED INCORRECTLY, OR A PERIOD IS MISSING',E,'OR IT HAS',
        ' NOT BEEN COMPLETED ON A SINGLE CARD',E,'THE DATA ',
        'VARIABLE MAY NOT HAVE BEEN FOLLOWED BY A LEFT ARROW',E,
        'ON THE SAME CARD AS ITSELF',E>);
        GO TO L198;
L186:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'THE FIRST CHARACTER OF THE ABOVE DATA VARIABLE IS NOT ',
        'A LEGAL LETTER',E>);
        GO TO L198;
L187:   PRINT(<120B,1C,'THE ABOVE CARD HAS AN IMPROPER DATA ',
        'VARIABLE',E,'IT IS PROBABLY SPELLED INCORRECTLY',E>);
        GO TO L198;
L188:   PRINT(<120B,1C,'THERE IS AN ERROR ON ONE OF THE ABOVE ',
        'TWO CARDS',E,'AN ILLEGAL PIECE OF DATA IS ',
        'LOCATED ON THE ABOVE CARD',E,'OR ELSE A SEMICOLON ',
        'SHOULD HAVE BEEN PLACED',E,'AT THE END OF DATA ',
        'ON ONE OF THE ABOVE TWO CARDS',E>);
        GO TO L198;
L189:   PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
        'BOTH ALPHANUMERIC AND NUMERIC DATA ',
        'ARE GIVEN TO THE SAME ARRAY',E>);
        GO TO L198;
```

```
    L190: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A NUMBER HAS MORE THAN ONE PLUS OR MINUS SIGN',E,
          'OR ELSE THE SIGN IS NOT IN FRONT OF THE NUMBER',E>);
          GO TO L198;
    L191: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A DATA CARD SHOULD END WITH EITHER A COMMA OR ',
          'A SEMICOLON',E>);
          GO TO L198;
    L192: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A PIECE OF NUMERICAL DATA HAS MORE THAN ONE ',
          'DECIMAL POINT',E>);
          GO TO L198;
    L193: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A PIECE OF NUMERICAL DATA WAS NOT FOLLOWED BY A ',
          'COMMA OR A SEMICOLON',E,'OR ELSE AN ILLEGAL PIECE ',
          'OF DATA WAS ENTERED',E,'OR ELSE MORE THAN ONE PIECE OF ',
          'DATA',E,'WAS GIVEN TO A NUMERICAL SINGLE VARIABLE',E>);
          GO TO L198;
    L194: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'BOTH NUMERIC AND ALPHANUMERIC DATA ',
          'ARE GIVEN TO THE SAME ARRAY',E>);
          GO TO L198;
    L195: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'THE END OF THE CARD WAS REACHED',E,
          'BEFORE A PIECE OF ALPHANUMERIC DATA WAS COMPLETED',E>);
          GO TO L198;
    L196: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A PIECE OF ALPHANUMERICAL DATA WAS NOT FOLLOWED BY A ',
          'COMMA OR SEMICOLON',E,'OR ELSE MORE THAN ONE PIECE OF ',
          'DATA',E,'WAS GIVEN TO AN ALPHANUMERIC SINGLE VARIABLE',E>);
          GO TO L198;
    L197: PRINT(<120B,1C,'THERE IS AN ERROR ON THE ABOVE CARD',E,
          'A PROCEDURE CALL HAS BEEN PLACED ON A DATA CARD',E,
          'PROCEDURE CALLS MUST OCCUPY CARDS BY THEMSELVES',E>);
    L198: ERROR ← TRUE;
          TRANSFER ← 1;
          SKIPOVER ← FALSE;
          I ← 80;
          GO TO L110;
    ERR1: PRINT(<'PHASE ONE COMPILER ERROR',E>);
    END;
  PHASE.ONE.RECOGNIZE;
  END;
  BEGIN
  PRINT(<'THE SEGMENT IS STORED',E>); .
  END;



  BEGIN
SY SEGMENT 53
SY RELEASE WHAT
SY RELEASE SYMBOLIC LIBRARY
```

A2.24

```
AL COMMENT   THE HYDRO COMPILER - PHASE TWO,SECTION ONE;
     OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
     STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
     PAIRS,TEMP1,TEMP2;
     OWN LOGIC TYPEDATA;
     OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
     OWN INTEGER ARRAY STACK[1:1000];
     OWN INTEGER ARRAY PROGRAM[1:500,1:2];
   PROCEDURE PHASE.TWO.ASSEMBLE;
     BEGIN
     INTEGER BASE1,BASE2,BASE3,BASE4;
     BOOLEAN DECLARED;
     LIBRARY PROCEDURE LINK;
     PROCEDURE MOVE.CARD (A,B);
        VALUE A,B;
        INTEGER A,B;
        BEGIN
        PAIRS ← PAIRS+1;
        PROGRAM[PAIRS,1] ← A;
        PROGRAM[PAIRS,2] ← B;
        END;
     BOOLEAN PROCEDURE IN.THE.STACK (A);
        VALUE A;
        INTEGER A;
        BEGIN
        IN.THE.STACK ← FALSE;
        FOR I ← OPNO-1 STEP -1 UNTIL STORBASE DO
           IF STACK[I] = A THEN BEGIN
               IN.THE.STACK ← TRUE;
               GO TO L200;   END;
        L200:   END;
     PROCEDURE CHECK.STACK (A);
        VALUE A;
        INTEGER A;
        BEGIN
        IF ¬IN.THE.STACK(A) THEN BEGIN
           ERROR ← TRUE;
           PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
           'A PROCEDURE HAS BEEN CALLED WITHOUT ALL ',
           'OF THE DATA REQUIRED FOR IT',E,'BEING ENTERED.  ',
           'THIS DATA CAN BE ENTERED ON DATA CARDS PLACED',E,
           'IMMEDIATELY FOLLOWING THE PROCEDURE CALL, OR BY ',
           'MEANS OF CALLS',E,'ON OTHER PROCEDURES WHICH ',
           'CALCULATE THE DATA.  YOU MAY ALSO HAVE',E,
           'NEGLECTED TO CALL A PROCEDURE WHICH IS REQUIRED ',
           'IN YOUR PROGRAM.',E>);
           END;
        END;
     PROCEDURE LOAD.STACK (A);
        VALUE A;
        INTEGER A;
        BEGIN
        STORBASE ← STORBASE-1;
        STACK[STORBASE] ← A;
        END;
```

```
PROCEDURE SINGLE.VARIABLE;
   BEGIN
   J ← STACK[OPNO]+BASE1+1;
   MOVE.CARD(J,J);
   END;
PROCEDURE ARRAY.VARIABLE;
   BEGIN
   PROCEDURE TEST.BOUNDS (A,B,C);
      VALUE A,B,C;
      INTEGER A,B,C;
      BEGIN
      IF IN.THE.STACK(A) ∧ IN.THE.STACK(B) ∧ IN.THE.STACK(C)
      THEN GO TO L240
      ELSE ERROR ← TRUE;
      PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
      'DATA ARE GIVEN FOR AN ARRAY BEFORE ONE OF THE ',
      'BOUNDS ON THAT ARRAY',E,'IS DEFINED.  ',
      'THE BOUND CAN BE DEFINED BY A DATA READ ABOVE ',
      'THAT OF',E,'THE ARRAY, OR BY A CALL ON A PROCEDURB ',
      'WHICH DEFINES IT.',E>);
      GO TO L250;
      END;
   SWITCH ARRY ← A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
      A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,
      A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,
      A31,A32,A33,A34,A35,A36,A37,A38,A39,A40,
      A41,A42,A43,A44,A45,A46,A47,A48,A49,A50,
      A51,A52,A53,A54,A55,A56;
   IF IN.THE.STACK(STACK[OPNO]) THEN DECLARED ← TRUE
      ELSE DECLARED ← FALSE;
   J ← STACK[OPNO] - ARRAYBASE;
   K ← BASE2+2+(J-1)*3;
   IF ¬DECLARED THEN G7 TO ARRY[J]
   ELSE GO TO L245;
      A1:   TEST.BOUNDS(43,0,0);
      A2:   TEST.BOUNDS(40,0,0);
      A3:   TEST.BOUNDS(29,0,0);
      A4:   TEST.BOUNDS(46,0,0);
      A5:   TEST.BOUNDS(45,0,0);
      A6:   TEST.BOUNDS(49,44,0);
      A7:   TEST.BOUNDS(12,11,0);
      A8:   TEST.BOUNDS(30,0,0);
      A9:   TEST.BOUNDS(40,0,0);
      A10:  TEST.BOUNDS(41,0,0);
      A11:  TEST.BOUNDS(42,0,0);
      A12:  TEST.BOUNDS(35,0,0);
      A13:  TEST.BOUNDS(38,37,0);
      A14:  TEST.BOUNDS(40,0,0);
      A15:  TEST.BOUNDS(41,0,0);
      A16:  TEST.BOUNDS(42,0,0);
      A17:  TEST.BOUNDS(12,11,0);
      A18:  TEST.BOUNDS(48,0,0);
      A19:  TEST.BOUNDS(45,0,0);
      A20:  TEST.BOUNDS(33,0,0);
      A21:  TEST.BOUNDS(33,0,0);
```

```
        A22:  TEST.BOUNDS(33,0,0);
        A23:  TEST.BOUNDS(43,0,0);
        A24:  TEST.BOUNDS(45,0,0);
        A25:  TEST.BOUNDS(30,0,0);
        A26:  TEST.BOUNDS(34,0,0);
        A27:  TEST.BOUNDS(32,0,0);
        A28:  TEST.BOUNDS(38,0,0);
        A29:  TEST.BOUNDS(41,0,0);
        A30:  TEST.BOUNDS(44,0,0);
        A31:  TEST.BOUNDS(44,0,0);
        A32:  TEST.BOUNDS(38,0,0);
        A33:  TEST.BOUNDS(41,0,0);
        A34:  TEST.BOUNDS(42,0,0);
        A35:  TEST.BOUNDS(50,53,54);
        A36:  TEST.BOUNDS(60,0,0);
        A37:  TEST.BOUNDS(50,53,54);
        A38:  TEST.BOUNDS(50,53,54);
        A39:  TEST.BOUNDS(50,53,54);
        A40:  TEST.BOUNDS(50,53,54);
        A41:  TEST.BOUNDS(61,0,0);
        A42:  TEST.BOUNDS(50,53,0);
        A43:  TEST.BOUNDS(50,53,0);
        A44:  TEST.BOUNDS(50,53,54);
        A45:  TEST.BOUNDS(50,53,55);
        A46:  TEST.BOUNDS(50,53,0);
        A47:  TEST.BOUNDS(50,53,0);
        A48:  TEST.BOUNDS(50,53,0);
        A49:  TEST.BOUNDS(50,53,0);
        A50:  TEST.BOUNDS(50,53,0);
        A51:  TEST.BOUNDS(61,0,0);
        A52:  TEST.BOUNDS(50,53,54);
        A53:  TEST.BOUNDS(50,53,55);
        A54:  TEST.BOUNDS(50,53,55);
        A55:  TEST.BOUNDS(50,53,54);
        A56:  TEST.BOUNDS(50,53,0);
   L240:   MOVE.CARD(K,K+2);    GO TO L250;
   L245:   MOVE.CARD(K+1,K+2);
   L250:   END;
PROCEDURE PROCESS.VARIABLES;
   BEGIN
   L206:   OPNO ← OPNO+1;
   IF STACK[OPNO] < ARRAYBASE THEN BEGIN
      SINGLE.VARIABLE;
      GO TO L206;   END;
   IF STACK[OPNO] < PROCBASE THEN BEGIN
      ARRAY.VARIABLE;
      GO TO L206;   END;
   END;
PROCEDURE PROCESS.VARIABLES.EXCEPT (A);
   VALUE A;
   INTEGER A;
   BEGIN
   L207:   OPNO ← OPNO+1;
   IF STACK[OPNO] = A THEN GO TO L208;
   IF STACK[OPNO] < ARRAYBASE THEN BEGIN
```

```
         SINGLE.VARIABLE;
         GO TO L207;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
         ARRAY.VARIABLE;
         GO TO L207;   END;
      L208:   OPNO ← OPNO+1;
         IF STACK[OPNO] < PROCBASE THEN GO TO L208;
      END;
   PROCEDURE PROCESS.VARIABLES.UNTIL (A);
      VALUE A;
      INTEGER A;
      BEGIN
      L209:   OPNO ← OPNO+1;
      IF STACK[OPNO] = A THEN GO TO L210;
      IF STACK[OPNO] < ARRAYBASE THEN BEGIN
         SINGLE.VARIABLE;
         GO TO L209;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
         ARRAY.VARIABLE;
         GO TO L209;   END;
      L210:   OPNO ← OPNO-1;
      END;
COMMENT   CODE FOR THE PHASE.TWO PROCEDURE;
   SWITCH PROC ← P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,
         P11,P12,P13,P14,P15,P16,P17,P18,P19,P20,
         P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,
         P31,P32,P33,P34,P35,P36,P37,P38,P39,P40,
         P41,P42,P43,P44,P45,P46,P47,P48,P49,P50,
         P51,P52,P53,P54,P55,P56,P57,P58,P59,P60,
         P61,P62,P63,P64,P65,P66,P67,P68,P69,P70,
         P71,P72,P73,P74,P75,P76,P77,P78,P79,P80,
         P81,P82,P83,P84,P85;
   BASE1 ← 262;
   BASE2 ← 327;
   BASE3 ← 497;
   BASE4 ← 696;
      IF ¬DATAREAD THEN BEGIN
         LOAD.STACK(0);
         MOVE.CARD(12,BASE1-1);
         OPNO ← OPNO-1;
         PROCESS.VARIABLES;
         DATAREAD ← TRUE;
         END;
   L275: STACK[TEMP1] ← TEMP2;
         J ← STACK[OPNO] - PROCBASE + 1;
         TEMP1 ← OPNO;
         TEMP2 ← STACK[OPNO];
         STACK[OPNO] ← 0;
         GO TO PROC[J];
   P0:   IF ¬ ERROR THEN BEGIN
            MOVE.CARD(1,6);
            LINK(57);
            END ELSE HALT;
   P1: P4: P5: P17: P18: P19: P20: P23: P24: P25: P27: P29:
   P30: P31: P33: P34: P41:          LINK(54);
```

```
 P46: P47: P48: P49: P50: P51: P52: P53: P54: P55:
 P56: P57: P58: P59:                   LINK(55);
 P60: P61: P62: P63: P64: P65: P66: P67: P68: P69:
 P70: P71: P72: P73: P74: P75: P76: P77: P78: P79: P80:  LINK(56)
 P2:    GO TO L275;
 P3:    PROCESS.VARIABLES;
        IF ¬ IN.THE.STACK(PROCBASE+3) THEN BEGIN
           CHECK.STACK(ARRAYBASE+25);
           MOVE.CARD(BASE4+42,BASE4+53);
        END ELSE MOVE.CARD(BASE4+53,BASE4+63);
        GO TO L275;
 P6:    PROCESS.VARIABLES;
        IF ¬ IN.THE.STACK(PROCBASE+6) THEN BEGIN
           CHECK.STACK(ARRAYBASE+25);
           LOAD.STACK(SPECBASE+20);
           IF ¬IN.THE.STACK(PROCBASE+11) THEN
           MOVE.CARD(BASE3+17,BASE3+19);
           MOVE.CARD(BASE4+161,BASE4+180);
        END ELSE MOVE.CARD(BASE4+180,BASE4+180);
        GO TO L275;
 P7:    PROCESS.VARIABLES;
        IF ¬ IN.THE.STACK(PROCBASE+7) THEN BEGIN
           CHECK.STACK(ARRAYBASE+25);
           LOAD.STACK(SPECBASE+23);
           IF ¬IN.THE.STACK(PROCBASE+9) ∧
              ¬IN.THE.STACK(PROCBASE+12) THEN
           MOVE.CARD(BASE3+20,BASE3+22);
           MOVE.CARD(BASE4+185,BASE4+205);
        END ELSE MOVE.CARD(BASE4+205,BASE4+205);
        GO TO L275;
 P8:    PROCESS.VARIABLES;
        IF ¬ IN.THE.STACK(PROCBASE+8) THEN BEGIN
           CHECK.STACK(ARRAYBASE+3);
           LOAD.STACK(30);
           LOAD.STACK(ARRAYBASE+25);
           IF ¬IN.THE.STACK(PROCBASE+15) ∧
              ¬IN.THE.STACK(PROCBASE+28) ∧
              ¬IN.THE.STACK(PROCBASE+42) THEN
           MOVE.CARD(BASE3+23,BASE3+26);
           IF ¬IN.THE.STACK(PROCBASE+3) THEN BEGIN
              MOVE.CARD(BASE4+42,BASE4+52);
              LOAD.STACK(PROCBASE+3);   END;
           MOVE.CARD(BASE4+210,BASE4+225);
        END ELSE MOVE.CARD(BASE4+225,BASE4+225);
        GO TO L275;
 P9:    PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+9) THEN BEGIN
           CHECK.STACK(29);
           CHECK.STACK(ARRAYBASE+25);
           LOAD.STACK([illegible]);
           [illegible]AD.STAC[illegible]
           [illegible].STAC[illegible]PECBASE+59);
           [illegible]D.STACK(SPECBASE+23);
           LOAD.STACK(SPECBASE+45);
           IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
```

```
            MOVE.CARD(BASE4+22,BASE4+34);
            LOAD.STACK(PROCBASE+2);    END;
         IF ¬IN.THE.STACK(PROCBASE+35) THEN BEGIN
            MOVE.CARD(BASE4+1763,BASE4+1775);
            LOAD.STACK(PROCBASE+35);    END;
         IF ¬IN.THE.STACK(PROCBASE+37) THEN
         MOVE.CARD(BASE3+27,BASE3+29);
         IF ¬IN.THE.STACK(PROCBASE+7) ∧
            ¬IN.THE.STACK(PROCBASE+12) THEN
         MOVE.CARD(BASE3+20,BASE3+22);
         MOVE.CARD(BASE4+231,BASE4+299);
      END ELSE MOVE.CARD(BASE4+299,BASE4+299);
      GO TO L275;
P10:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+10) THEN BEGIN
         CHECK.STACK(14);
         CHECK.STACK(ARRAYBASE+25);
         LOAD.STACK(31);
         LOAD.STACK(ARRAYBASE+8);
         MOVE.CARD(BASE3+30,BASE3+32);
         MOVE.CARD(BASE4+304,BASE4+338);
      END ELSE MOVE.CARD(BASE4+338,BASE4+338);
      GO TO L275;
P11:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+11) THEN BEGIN
         CHECK.STACK(14);
         CHECK.STACK(ARRAYBASE+25);
         LOAD.STACK(31);
         LOAD.STACK(ARRAYBASE+8);
         LOAD.STACK(SPECBASE+20);
         IF ¬IN.THE.STACK(PROCBASE+10) THEN BEGIN
            MOVE.CARD(BASE3+30,BASE3+32);
            MOVE.CARD(BASE4+304,BASE4+337);
            LOAD.STACK(PROCBASE+10);    END;
         IF ¬IN.THE.STACK(PROCBASE+6) THEN
         MOVE.CARD(BASE3+17,BASE3+19);
         MOVE.CARD(BASE4+343,BASE4+366);
      END ELSE MOVE.CARD(BASE4+366,BASE4+366);
      GO TO L275;
P12:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+12) THEN BEGIN
         CHECK.STACK(14);
         CHECK.STACK(ARRAYBASE+25);
         LOAD.STACK(31);
         LOAD.STACK(ARRAYBASE+8);
         LOAD.STACK(SPECBASE+23);
         IF ¬IN.THE.STACK(PROCBASE+10) THEN BEGIN
            MOVE.CARD(BASE3+30,BASE3+32);
            MOVE.CARD(BASE4+304,BASE4+337);
            LOAD.STACK(PROCBASE+10);    END;
         IF ¬IN.THE.STACK(PROCBASE+7) ∧
            ¬IN.THE.STACK(PROCBASE+9) THEN
         MOVE.CARD(BASE3+20,BASE3+22);
         MOVE.CARD(BASE4+371,BASE4+399);
      END ELSE MOVE.CARD(BASE4+399,BASE4+399);
```

```
        GO TO L275;
  P13:  PROCESS.VARIABLES;
           MOVE.CARD(BASE3+33,BASE3+36);
        IF ¬IN.THE.STACK(PROCBASE+13) THEN BEGIN
           IF ¬IN.THE.STACK(PROCBASE+38) ∧
              ¬IN.THE.STACK(PROCBASE+39) ∧
              ¬IN.THE.STACK(PROCBASE+40) ∧
              ¬IN.THE.STACK(PROCBASE+43) THEN BEGIN
              CHECK.STACK(10);
              CHECK.STACK(15);
              CHECK.STACK(ARRAYBASE+31);   END;
           CHECK.STACK(44);
           LOAD.STACK(47);
           LOAD.STACK(SPECBASE+24);
           MOVE.CARD(BASE4+403,BASE4+436);
        END ELSE MOVE.CARD(BASE4+436,BASE4+436);
        GO TO L275;
  P14:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+14) THEN BEGIN
           IF ¬IN.THE.STACK(PROCBASE+26) THEN BEGIN
              CHECK.STACK(15);
              CHECK.STACK(50);
              CHECK.STACK(ARRAYBASE+30);
              CHECK.STACK(ARRAYBASE+31);   END;
           CHECK.STACK(ARRAYBASE+5);
           LOAD.STACK(SPECBASE+25);
           MOVE.CARD(BASE3+37,BASE3+39);
           MOVE.CARD(BASE4+443,BASE4+480);
        END ELSE MOVE.CARD(BASE4+480,BASE4+480);
        GO TO L275;
  P15:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+15) THEN BEGIN
           CHECK.STACK(20);
           CHECK.STACK(19);
           CHECK.STACK(ARRAYBASE+3);
           LOAD.STACK(30);
           LOAD.STACK(4);
           LOAD.STACK(63);
           LOAD.STACK(ARRAYBASE+25);
           IF ¬IN.THE.STACK(PROCBASE+8) ∧
              ¬IN.THE.STACK(PROCBASE+28) ∧
              ¬IN.THE.STACK(PROCBASE+42) THEN
           MOVE.CARD(BASE3+23,BASE3+26);
           IF ¬IN.THE.STACK(PROCBASE+3) THEN BEGIN
              MOVE.CARD(BASE4+42,BASE4+52);
              LOAD.STACK(PROCBASE+3);   END;
           IF ¬IN.THE.STACK(PROCBASE+28) THEN BEGIN
              MOVE.CARD(BASE4+1404,BASE4+1430);
              LOAD.STACK(PROCBASE+28);   END;
           IF ¬IN.THE.STACK(PROCBASE+42) THEN BEGIN
              MOVE.CARD(BASE4+2013,BASE4+2065);
              LOAD.STACK(PROCBASE+42);   END;
           MOVE.CARD(BASE4+497,BASE4+499);
        END ELSE MOVE.CARD(BASE4+499,BASE4+499);
        GO TO L275;
```

```
P16:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+16) THEN BEGIN
         LOAD.STACK(R1);
         MOVE.CARD(BASE4+505,BASE4+515);
      END ELSE MOVE.CARD(BASE4+515,BASE4+515);
      GO TO L275;
P21:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+21) THEN BEGIN
         CHECK.STACK(ARRAYBASE+25);
         LOAD.STACK(22);
         IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
            MOVE.CARD(BASE4+22,BASE4+34);
            LOAD.STACK(PROCBASE+2);   END;
         MOVE.CARD(BASE4+1139,BASE4+1149);
      END ELSE MOVE.CARD(BASE4+1149,BASE4+1149);
      GO TO L275;
P22:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+22) THEN BEGIN
         LOAD.STACK(R2);
         MOVE.CARD(BASE4+1156,BASE4+1166);
      END ELSE MOVE.CARD(BASE4+1165,BASE4+1166);
      GO TO L275;
P26:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+26) THEN BEGIN
         CHECK.STACK(2);
         CHECK.STACK(15);
         CHECK.STACK(ARRAYBASE+19);
         CHECK.STACK(ARRAYBASE+24);
         CHECK.STACK(ARRAYBASE+30);
         LOAD.STACK(ARRAYBASE+31);
         MOVE.CARD(BASE3+111,BASE3+113);
         MOVE.CARD(BASE4+1260,BASE4+1379);
      END ELSE MOVE.CARD(BASE4+1379,BASE4+1379);
      GO TO L275;
P28:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+28) THEN BEGIN
         CHECK.STACK(20);
         CHECK.STACK(4);
         CHECK.STACK(ARRAYBASE+3);
         LOAD.STACK(30);
         LOAD.STACK(63);
         LOAD.STACK(ARRAYBASE+25);
         IF ¬IN.THE.STACK(PROCBASE+8) ∧
            ¬IN.THE.STACK(PROCBASE+15) ∧
            ¬IN.THE.STACK(PROCBASE+42) THEN
         MOVE.CARD(BASE3+23,BASE3+26);
         IF ¬IN.THE.STACK(PROCBASE+3) THEN BEGIN
            MOVE.CARD(BASE4+42,BASE4+62);
            LOAD.STACK(PROCBASE+3);   END;
         MOVE.CARD(BASE4+1404,BASE4+1431);
      END ELSE MOVE.CARD(BASE4+1431,BASE4+1431);
      GO TO L275;
P32:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+32) THEN BEGIN
         CHECK.STACK(9);
```

```
            CHECK.STACK(10);
            CHECK.STACK(15);
            CHECK.STACK(ARRAYBASE+31);
            MOVE.CARD(BASE4+1587,BASE4+1667);
         END ELSE MOVE.CARD(BASE4+1667,BASE4+1667);
         GO TO L275;
   P35:  GO TO L275;
   P36:  PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+36) THEN BEGIN
            CHECK.STACK(22);
            CHECK.STACK(ARRAYBASE+25);
            LOAD.STACK(8);
            IF ¬IN.THE.STACK(PROCBASE+35) THEN BEGIN
               MOVE.CARD(BASE4+1763,BASE4+1775);
               LOAD.STACK(PROCBASE+35);   END;
            MOVE.CARD(BASE4+1783,BASE4+1793);
         END ELSE MOVE.CARD(BASE4+1793,BASE4+1793);
         GO TO L275;
   P37:  PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+37) THEN BEGIN
            CHECK.STACK(20);
            CHECK.STACK(24);
            CHECK.STACK(63);
            CHECK.STACK(ARRAYBASE+25);
            LOAD.STACK(SPECBASE+45);
            IF ¬IN.THE.STACK(PROCBASE+9) THEN
            MOVE.CARD(BASE3+27,BASE3+29);
            MOVE.CARD(BASE4+1800,BASE4+1844);
         END ELSE MOVE.CARD(BASE4+1844,BASE4+1844);
         GO TO L275;
   P38:  PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+38) THEN BEGIN
            IF ¬IN.THE.STACK(PROCBASE+39) THEN BEGIN
               CHECK.STACK(2);
               CHECK.STACK(15);
               CHECK.STACK(ARRAYBASE+1);
               CHECK.STACK(ARRAYBASE+18);
               CHECK.STACK(ARRAYBASE+23);
               LOAD.STACK(10);
               LOAD.STACK(44);
               LOAD.STACK(ARRAYBASE+31);
               LOAD.STACK(SPECBASE+27);
               MOVE.CARD(BASE3+127,BASE3+137);   END;
            MOVE.CARD(BASE4+1849,BASE4+1882);
         END ELSE MOVE.CARD(BASE4+1882,BASE4+1882);
         GO TO L275;
   P39:  PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+39) THEN BEGIN
            IF ¬IN.THE.STACK(PROCBASE+38) THEN BEGIN
               CHECK.STACK(2);
               CHECK.STACK(15);
               CHECK.STACK(ARRAYBASE+1);
               CHECK.STACK(ARRAYBASE+18);
               CHECK.STACK(ARRAYBASE+23);
               LOAD.STACK(10);
            LOAD.STACK(44);
            LOAD.STACK(ARRAYBASE+31);
            LOAD.STACK(SPECBASE+27);
            MOVE.CARD(BASE3+127,BASE3+137);   END;
         CHECK.STACK(23);
         MOVE.CARD(BASE4+1889,BASE4+1924);
      END ELSE MOVE.CARD(BASE4+1924,BASE4+1924);
      GO TO L275;
P40:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+40) THEN BEGIN
         IF ¬IN.THE.STACK(PROCBASE+44) THEN BEGIN
            CHECK.STACK(21);
            CHECK.STACK(24);
            CHECK.STACK(25);   END;
         CHECK.STACK(3);
         CHECK.STACK(10);
         CHECK.STACK(24);
         CHECK.STACK(27);
         LOAD.STACK(44);
         LOAD.STACK(ARRAYBASE+31);
         MOVE.CARD(BASE3+138,BASE3+141);
         MOVE.CARD(BASE4+1937,BASE4+1974);
      END ELSE MOVE.CARD(BASE4+1974,BASE4+1974);
      GO TO L275;
P42:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+42) THEN BEGIN
         CHECK.STACK(20);
         CHECK.STACK(19);
         CHECK.STACK(ARRAYBASE+3);
         LOAD.STACK(30);
         LOAD.STACK(63);
         LOAD.STACK(ARRAYBASE+25);
         IF ¬IN.THE.STACK(PROCBASE+8) ^
            ¬IN.THE.STACK(PROCBASE+15) ^
            ¬IN.THE.STACK(PROCBASE+28) THEN
         MOVE.CARD(BASE3+23,BASE3+26);
         IF ¬IN.THE.STACK(PROCBASE+3) THEN BEGIN
            MOVE.CARD(BASE4+42,BASE4+62);
            LOAD.STACK(PROCBASE+3);   END;
         MOVE.CARD(BASE4+2018,BASE4+2066);
      END ELSE MOVE.CARD(BASE4+2065,BASE4+2066);
      GO TO L275;
P43:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+43) THEN BEGIN
         CHECK.STACK(2);
         CHECK.STACK(5);
         CHECK.STACK(10);
         CHECK.STACK(15);
         LOAD.STACK(44);
         LOAD.STACK(ARRAYBASE+31);
         MOVE.CARD(BASE3+148,BASE3+152);
         MOVE.CARD(BASE4+2071,BASE4+2094);
      END ELSE MOVE.CARD(BASE4+2094,BASE4+2094);
      GO TO L275;
P44:  PROCESS.VARIABLES;
```

```
          IF ¬IN.THE.STACK(PROCBASE+44) THEN BEGIN
             CHECK.STACK(ARRAYBASE+5);
             LOAD.STACK(SPECBASE+19);
             LOAD.STACK(21);
             LOAD.STACK(24);
             LOAD.STACK(25);
             MOVE.CARD(BASE3+153,BASE3+155);
             MOVE.CARD(BASE4+2101,BASE4+2181);
          END ELSE MOVE.CARD(BASE4+2181,BASE4+2181);
          GO TO L275;
    P45:  GO TO L275;
    P81:  GO TO L275;
    P82:  PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+82) THEN BEGIN
             IF ¬IN.THE.STACK(PROCBASE+81) THEN BEGIN
                MOVE.CARD(BASE4+3683,BASE4+3815);
                LOAD.STACK(PROCBASE+81);   END;
             MOVE.CARD(BASE4+3823,BASE4+3828);
          END ELSE MOVE.CARD(BASE4+3829,BASE4+3828);
          GO TO L275;
    P83:  PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+83) THEN BEGIN
             MOVE.CARD(BASE4+3836,BASE4+3841);
          END ELSE MOVE.CARD(BASE4+3841,BASE4+3841);
          GO TO L275;
    P84:  PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+84) THEN BEGIN
             MOVE.CARD(BASE4+3849,BASE4+3854);
          END ELSE MOVE.CARD(BASE4+3854,BASE4+3854);
          GO TO L275;
    P85:  PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+85) THEN BEGIN
             MOVE.CARD(BASE4+3862,BASE4+3867);
          END ELSE MOVE.CARD(BASE4+3867,BASE4+3867);
          GO TO L275;
    END;
  PHASE.TWO.ASSEMBLE;
  END;
  BEGIN
  PRINT(<'THE SEGMENT IS STORED',E>);
  END;




   BEGIN
SY SEGMENT 54
SY RELEASE WHAT
SY RELEASE SYMBOLIC LIBRARY
AL COMMENT   THE HYDRO COMPILER - PHASE TWO,SECTION TWO;
      OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
      STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
      PAIRS,TEMP1,TEMP2;
      OWN LOGIC TYPEDATA;
      OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
      OWN INTEGER ARRAY STACK[1:1000];
      OWN INTEGER ARRAY PROGRAM[1:500,1:2];
   PROCEDURE PHASE.TWO.ASSEMBLE;
      BEGIN
      INTEGER BASE1,BASE2,BASE3,BASE4;
      BOOLEAN DECLARED;
      LIBRARY PROCEDURE LINK;
      PROCEDURE MOVE.CARD (A,B);
         VALUE A,B;
         INTEGER A,B;
         BEGIN
         PAIRS ← PAIRS+1;
         PROGRAM[PAIRS,1] ← A;
         PROGRAM[PAIRS,2] ← B;
         END;
      BOOLEAN PROCEDURE IN.THE.STACK (A);
         VALUE A;
         INTEGER A;
         BEGIN
         IN.THE.STACK ← FALSE;
         FOR I ← OPNO-1 STEP -1 UNTIL STORBASE DO
            IF STACK[I] = A THEN BEGIN
                 IN.THE.STACK ← TRUE;
                 GO TO L200;    END;
         L200:    END;
      PROCEDURE CHECK.STACK (A);
         VALUE A;
         INTEGER A;
         BEGIN
         IF ¬IN.THE.STACK(A) THEN BEGIN
            ERROR ← TRUE;
            PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
            'A PROCEDURE HAS BEEN CALLED WITHOUT ALL ',
            'OF THE DATA REQUIRED FOR IT',E,'BEING ENTERED.  ',
            'THIS DATA CAN BE ENTERED ON DATA CARDS PLACED',E,
            'IMMEDIATELY FOLLOWING THE PROCEDURE CALL, OR BY ',
            'MEANS OF CALLS',E,'ON OTHER PROCEDURES WHICH ',
            'CALCULATE THE DATA.  YOU MAY ALSO HAVE',E,
            'NEGLECTED TO CALL A PROCEDURE WHICH IS REQUIRED ',
            'IN YOUR PROGRAM.',E>);
            END;
         END;
      PROCEDURE LOAD.STACK (A);
         VALUE A;
         INTEGER A;
         BEGIN
         STORBASE ← STORBASE-1;
         STACK[STORBASE] ← A;
         END;
      PROCEDURE SINGLE.VARIABLE;
         BEGIN
         J ← STACK[OPNO]+BASE1+1;
         MOVE.CARD(J,J);
         END;
```

```
PROCEDURE ARRAY.VARIABLE;
   BEGIN
   PROCEDURE TEST.BOUNDS (A,B,C);
      VALUE A,B,C;
      INTEGER A,B,C;
      BEGIN
      IF IN.THE.STACK(A) ∧ IN.THE.STACK(B) ∧ IN.THE.STACK(C)
      THEN GO TO L240
      ELSE ERROR ← TRUE;
      PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
      'DATA ARE GIVEN FOR AN ARRAY BEFORE ONE OF THE ',
      'BOUNDS ON THAT ARRAY',E,'IS DEFINED.  ',
      'THE BOUND CAN BE DEFINED BY A DATA READ ABOVE ',
      'THAT OF',E,'THE ARRAY, OR BY A CALL ON A PROCEDURE ',
      'WHICH DEFINES IT.',E>);
      GO TO L250;
      END;
   SWITCH ARRY ← A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
      A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,
      A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,
      A31,A32,A33,A34,A35,A36,A37,A38,A39,A40,
      A41,A42,A43,A44,A45,A46,A47,A48,A49,A50,
      A51,A52,A53,A54,A55,A56;
   IF IN.THE.STACK(STACK[OPNO]) THEN DECLARED ← TRUE
      ELSE DECLARED ← FALSE;
   J ← STACK[OPNO] - ARRAYBASE;
   K ← BASE2+2+(J-1)*3;
   IF ¬DECLARED THEN GO TO ARRY[J]
   ELSE GO TO L245;
      A1:    TEST.BOUNDS(43,0,0);
      A2:    TEST.BOUNDS(40,0,0);
      A3:    TEST.BOUNDS(29,0,0);
      A4:    TEST.BOUNDS(46,0,0);
      A5:    TEST.BOUNDS(45,0,0);
      A6:    TEST.BOUNDS(49,44,0);
      A7:    TEST.BOUNDS(12,11,0);
      A8:    TEST.BOUNDS(30,0,0);
      A9:    TEST.BOUNDS(40,0,0);
      A10:   TEST.BOUNDS(41,0,0);
      A11:   TEST.BOUNDS(42,0,0);
      A12:   TEST.BOUNDS(35,0,0);
      A13:   TEST.BOUNDS(36,37,0);
      A14:   TEST.BOUNDS(40,0,0);
      A15:   TEST.BOUNDS(41,0,0);
      A16:   TEST.BOUNDS(42,0,0);
      A17:   TEST.BOUNDS(12,11,0);
      A18:   TEST.BOUNDS(48,0,0);
      A19:   TEST.BOUNDS(45,0,0);
      A20:   TEST.BOUNDS(33,0,0);
      A21:   TEST.BOUNDS(33,0,0);
      A22:   TEST.BOUNDS(33,0,0);
      A23:   TEST.BOUNDS(43,0,0);
      A24:   TEST.BOUNDS(45,0,0);
      A25:   TEST.BOUNDS(36,0,0);
      A26:   TEST.BOUNDS(34,0,0);
```

```
          A27:  TEST.BOUNDS(32,0,0);
          A28:  TEST.BOUNDS(38,0,0);
          A29:  TEST.BOUNDS(41,0,0);
          A30:  TEST.BOUNDS(44,0,0);
          A31:  TEST.BOUNDS(44,0,0);
          A32:  TEST.BOUNDS(38,0,0);
          A33:  TEST.BOUNDS(41,0,0);
          A34:  TEST.BOUNDS(42,0,0);
          A35:  TEST.BOUNDS(50,53,54);
          A36:  TEST.BOUNDS(60,0,0);
          A37:  TEST.BOUNDS(50,53,54);
          A38:  TEST.BOUNDS(50,53,54);
          A39:  TEST.BOUNDS(50,53,54);
          A40:  TEST.BOUNDS(50,53,54);
          A41:  TEST.BOUNDS(61,0,0);
          A42:  TEST.BOUNDS(50,53,0);
          A43:  TEST.BOUNDS(50,53,0);
          A44:  TEST.BOUNDS(50,53,54);
          A45:  TEST.BOUNDS(50,53,55);
          A46:  TEST.BOUNDS(50,53,0);
          A47:  TEST.BOUNDS(50,53,0);
          A48:  TEST.BOUNDS(50,53,0);
          A49:  TEST.BOUNDS(50,53,0);
          A50:  TEST.BOUNDS(50,53,0);
          A51:  TEST.BOUNDS(61,0,0);
          A52:  TEST.BOUNDS(50,53,54);
          A53:  TEST.BOUNDS(50,53,55);
          A54:  TEST.BOUNDS(50,53,55);
          A55:  TEST.BOUNDS(50,53,54);
          A56:  TEST.BOUNDS(50,53,0);
     L240:   MOVE.CARD(K,K+2);    GO TO L250;
     L245:   MOVE.CARD(K+1,K+2);
     L250:   END;
PROCEDURE PROCESS.VARIABLES;
     BEGIN
     L206:   OPNO ← OPNO+1;
     IF STACK[OPNO] < ARRAYBASE THEN BEGIN
          SINGLE.VARIABLE;
          GO TO L206;    END;
     IF STACK[OPNO] < PROCBASE THEN BEGIN
          ARRAY.VARIABLE;
          GO TO L206;    END;
     END;
PROCEDURE PROCESS.VARIABLES.EXCEPT (A);
     VALUE A;
     INTEGER A;
     BEGIN
     L207:   OPNO ← OPNO+1;
     IF STACK[OPNO] = A THEN GO TO L208;
     IF STACK[OPNO] < ARRAYBASE THEN BEGIN
          SINGLE.VARIABLE;
          GO TO L207;    END;
     IF STACK[OPNO] < PROCBASE THEN BEGIN
          ARRAY.VARIABLE;
          GO TO L207;    END;
```

```
      L208:   OPNO ← OPNO+1;
          IF STACK[OPNO] < PROCBASE THEN GO TO L208;
      END;
   PROCEDURE PROCESS.VARIABLES.UNTIL (A);
      VALUE A;
      INTEGER A;
      BEGIN
      L209:   OPNO ← OPNO+1;
      IF STACK[OPNO] = A THEN GO TO L210;
      IF STACK[OPNO] < ARRAYBASE THEN BEGIN
          SINGLE.VARIABLE;
          GO TO L209;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
          ARRAY.VARIABLE;
          GO TO L209;   END;
      L210:   OPNO ← OPNO-1;
      END;
COMMENT   CODE FOR THE PHASE.TWO PROCEDURE;
   SWITCH PROC ← P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,
          P11,P12,P13,P14,P15,P16,P17,P18,P19,P20,
          P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,
          P31,P32,P33,P34,P35,P36,P37,P38,P39,P40,
          P41,P42,P43,P44,P45,P46,P47,P48,P49,P50,
          P51,P52,P53,P54,P55,P56,P57,P58,P59,P60,
          P61,P62,P63,P64,P65,P66,P67,P68,P69,P70,
          P71,P72,P73,P74,P75,P76,P77,P78,P79,P80,
          P81,P82,P83,P84,P85;
   BASE1 ← 262;
   BASE2 ← 327;
   BASE3 ← 497;
   BASE4 ← 696;
   L275: STACK[TEMP1] ← TEMP2;
          J ← STACK[OPNO] - PROCBASE + 1;
          TEMP1 ← OPNO;
          TEMP2 ← STACK[OPNO];
          STACK[OPNO] ← 0;
          GO TO PROC[J];
   P0:    IF ¬ ERROR THEN BEGIN
              MOVE.CARD(1,6);
              LINK(57);
              END ELSE HALT;
   P2: P3: P6: P7: P8: P9: P10: P11: P12: P13: P14: P15: P16:
   P21: P22: P26: P28: P32: P35: P36: P37: P38: P39: P40:
   P42: P43: P44: P45: P81: P82: P83: P84: P85:      LINK(53);
   P46: P47: P48: P49: P50: P51: P52: P53: P54: P55:
   P56: P57: P58: P59:                  LINK(55);
   P60: P61: P62: P63: P64: P65: P66: P67: P68: P69:
   P70: P71: P72: P73: P74: P75: P76: P77: P78: P79: P80:  LINK(56);
   P1:    PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+1) THEN BEGIN
              CHECK.STACK(ARRAYBASE+26);
              LOAD.STACK(SPECBASE+42);
              IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
                  MOVE.CARD(BASE4+22,BASE4+34);
                  LOAD.STACK(PROCBASE+2);   END;
      MOVE.CARD(BASE4+3,BASE4+15);
   END ELSE MOVE.CARD(BASE4+16,BASE4+16);
   GO TO L275;
P4:   PROCESS.VARIABLES;
   IF ¬IN.THE.STACK(PROCBASE+4) THEN BEGIN
      CHECK.STACK(7);
      CHECK.STACK(ARRAYBASE+2);
      CHECK.STACK(ARRAYBASE+9);
      CHECK.STACK(ARRAYBASE+14);
      LOAD.STACK(SPECBASE+8);
      LOAD.STACK(SPECBASE+37);
      LOAD.STACK(SPECBASE+28);
      LOAD.STACK(SPECBASE+26);
      LOAD.STACK(SPECBASE+1);
      LOAD.STACK(SPECBASE+12);
      MOVE.CARD(BASE3+2,BASE3+9);
      MOVE.CARD(BASE4+68,BASE4+134);
   END ELSE MOVE.CARD(BASE4+134,BASE4+134);
   GO TO L275;
P5:   PROCESS.VARIABLES.EXCEPT(ARRAYBASE+12);
   IF ¬IN.THE.STACK(PROCBASE+5) THEN BEGIN
      CHECK.STACK(19);
      CHECK.STACK(35);
      CHECK.STACK(36);
      LOAD.STACK(SPECBASE+16);
      LOAD.STACK(SPECBASE+40);
      LOAD.STACK(SPECBASE+41);
      LOAD.STACK(SPECBASE+51);
      LOAD.STACK(SPECBASE+52);
      IF ¬IN.THE.STACK(PROCBASE+30) THEN BEGIN
         MOVE.CARD(BASE3+117,BASE3+119);
         MOVE.CARD(BASE4+1530,BASE4+1542);
         LOAD.STACK(PROCBASE+30);    END;
      IF ¬IN.THE.STACK(PROCBASE+45) THEN BEGIN
         MOVE.CARD(BASE4+2194,BASE4+2204);
         LOAD.STACK(PROCBASE+45);    END;
      IF ¬IN.THE.STACK(PROCBASE+34) THEN BEGIN
         MOVE.CARD(BASE4+1725,BASE4+1757);
         LOAD.STACK(PROCBASE+34);    END;
      MOVE.CARD(BASE3+10,BASE3+16);
      MOVE.CARD(BASE4+141,BASE4+146);
      MOVE.CARD(BASE2+36,BASE2+37);
      MOVE.CARD(BASE4+147,BASE4+150);
      MOVE.CARD(BASE2+36,BASE2+37);
      MOVE.CARD(BASE4+151,BASE4+154);
   END ELSE MOVE.CARD(BASE4+154,BASE4+154);
   GO TO L275;
P17:  PROCESS.VARIABLES;
   IF ¬IN.THE.STACK(PROCBASE+17) THEN BEGIN
      CHECK.STACK(PROCBASE+33);
      CHECK.STACK(PROCBASE+20);
      CHECK.STACK(PROCBASE+4);
      CHECK.STACK(PROCBASE+19);
      CHECK.STACK(PROCBASE+18);
      LOAD.STACK(SPECBASE+49);
```

```
            LOAD.STACK(SPECBASE+11);
            LOAD.STACK(SPECBASE+35);
            MOVE.CARD(BASE3+40,BASE3+45);
            MOVE.CARD(BASE4+521,BASE4+585);
         END ELSE MOVE.CARD(BASE4+585,BASE4+585);
         GO TO L275;
  P18:   PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+18) THEN BEGIN
            CHECK.STACK(PROCBASE+33);
            CHECK.STACK(PROCBASE+20);
            CHECK.STACK(PROCBASE+4);
            CHECK.STACK(PROCBASE+19);
            CHECK.STACK(ARRAYBASE+34);
            CHECK.STACK(ARRAYBASE+11);
            CHECK.STACK(ARRAYBASE+16);
            LOAD.STACK(SPECBASE+48);
            LOAD.STACK(SPECBASE+31);
            LOAD.STACK(SPECBASE+14);
            LOAD.STACK(SPECBASE+10);
            LOAD.STACK(SPECBASE+54);
            LOAD.STACK(SPECBASE+22);
            LOAD.STACK(SPECBASE+39);
            LOAD.STACK(SPECBASE+56);
            LOAD.STACK(SPECBASE+3);
            LOAD.STACK(SPECBASE+18);
            LOAD.STACK(SPECBASE+34);
            MOVE.CARD(BASE3+46,BASE3+63);
            MOVE.CARD(BASE4+590,BASE4+847);
         END ELSE MOVE.CARD(BASE4+847,BASE4+847);
         GO TO L275;
  P19:   PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+19) THEN BEGIN
            CHECK.STACK(PROCBASE+33);
            CHECK.STACK(PROCBASE+20);
            CHECK.STACK(PROCBASE+4);
            CHECK.STACK(ARRAYBASE+33);
            CHECK.STACK(ARRAYBASE+29);
            CHECK.STACK(ARRAYBASE+10);
            CHECK.STACK(ARRAYBASE+15);
            LOAD.STACK(SPECBASE+46);
            LOAD.STACK(SPECBASE+47);
            LOAD.STACK(SPECBASE+53);
            LOAD.STACK(SPECBASE+30);
            LOAD.STACK(SPECBASE+21);
            LOAD.STACK(SPECBASE+60);
            LOAD.STACK(SPECBASE+38);
            LOAD.STACK(SPECBASE+13);
            LOAD.STACK(SPECBASE+9);
            LOAD.STACK(SPECBASE+57);
            LOAD.STACK(SPECBASE+55);
            LOAD.STACK(SPECBASE+2);
            LOAD.STACK(SPECBASE+17);
            LOAD.STACK(SPECBASE+33);
            LOAD.STACK(SPECBASE+58);
            MOVE.CARD(BASE3+64,BASE3+103);
            MOVE.CARD(BASE4+853,BASE4+1094);
         END ELSE MOVE.CARD(BASE4+1094,BASE4+1094);
         GO TO L275;
P20:     PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+20) THEN BEGIN
            CHECK.STACK(ARRAYBASE+32);
            CHECK.STACK(ARRAYBASE+26);
            CHECK.STACK(ARRAYBASE+13);
            LOAD.STACK(39);
            LOAD.STACK(SPECBASE+32);
            MOVE.CARD(BASE3+104,BASE3+107);
            MOVE.CARD(BASE4+1100,BASE4+1133);
         END ELSE MOVE.CARD(BASE4+1133,BASE4+1133);
         GO TO L275;
P23:     PROCESS.VARIABLES.EXCEPT (ARRAYBASE+27);
         IF ¬IN.THE.STACK(PROCBASE+23) THEN BEGIN
            CHECK.STACK(32);
            CHECK.STACK(ARRAYBASE+20);
            CHECK.STACK(ARRAYBASE+21);
            CHECK.STACK(ARRAYBASE+22);
            LOAD.STACK(SPECBASE+4);
            LOAD.STACK(SPECBASE+5);
            LOAD.STACK(SPECBASE+6);
            LOAD.STACK(SPECBASE+7);
            LOAD.STACK(SPECBASE+44);
            IF ¬IN.THE.STACK(PROCBASE+27) THEN BEGIN
               IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
                  MOVE.CARD(BASE4+22,BASE4+34);
                  LOAD.STACK(PROCBASE+2);    END;
               MOVE.CARD(BASE3+114,BASE3+116);
               MOVE.CARD(BASE4+1385,BASE4+1396);
               LOAD.STACK(PROCBASE+27);    END;
            IF ¬IN.THE.STACK(PROCBASE+24) ∧
               ¬IN.THE.STACK(PROCBASE+25) THEN
            MOVE.CARD(BASE3+108,BASE3+110);
            MOVE.CARD(BASE4+1172,BASE4+1177);
            MOVE.CARD(BASE2+81,BASE2+92);
            MOVE.CARD(BASE4+1178,BASE4+1208);
         END ELSE MOVE.CARD(BASE4+1208,BASE4+1208);
         GO TO L275;
P24:     PROCESS.VARIABLES.EXCEPT (ARRAYBASE+27);
         IF ¬IN.THE.STACK(PROCBASE+24) THEN BEGIN
            CHECK.STACK(32);
            CHECK.STACK(ARRAYBASE+20);
            CHECK.STACK(ARRAYBASE+21);
            CHECK.STACK(ARRAYBASE+22);
            LOAD.STACK(SPECBASE+4);
            LOAD.STACK(SPECBASE+5);
            LOAD.STACK(SPECBASE+6);
            LOAD.STACK(SPECBASE+7);
            LOAD.STACK(SPECBASE+44);
            IF ¬IN.THE.STACK(PROCBASE+27) THEN BEGIN
               IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
                  MOVE.CARD(BASE4+22,BASE4+34);
                  LOAD.STACK(PROCBASE+2);    END;
```

```
              MOVE.CARD(BASE3+114,BASE3+116);
              MOVE.CARD(BASE4+1385,BASE4+1396);
              LOAD.STACK(PROCBASE+27);    END;
           IF ¬IN.THE.STACK(PROCBASE+23) ∧
              ¬IN.THE.STACK(PROCBASE+25) THEN
           MOVE.CARD(BASE3+109,BASE3+110);
           MOVE.CARD(BASE4+1212,BASE4+1218);
           MOVE.CARD(BASE2+81,BASE2+82);
           MOVE.CARD(BASE4+1219,BASE4+1235);
        END ELSE MOVE.CARD(BASE4+1235,BASE4+1235);
        GO TO L275;
P25:    PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+25) THEN BEGIN
           CHECK.STACK(ARRAYBASE+20);
           CHECK.STACK(ARRAYBASE+21);
           CHECK.STACK(ARRAYBASE+22);
           LOAD.STACK(SPECBASE+44);
           IF ¬IN.THE.STACK(PROCBASE+23) ∧
              ¬IN.THE.STACK(PROCBASE+24) THEN
           MOVE.CARD(BASE3+108,BASE3+110);
           MOVE.CARD(BASE4+1240,BASE4+1254);
        END ELSE MOVE.CARD(BASE4+1254,BASE4+1254);
        GO TO L275;
P27:    PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+27) THEN BEGIN
           CHECK.STACK(ARRAYBASE+27);
           LOAD.STACK(1);
           IF ¬IN.THE.STACK(PROCBASE+2) THEN BEGIN
              MOVE.CARD(BASE4+22,BASE4+34);
              LOAD.STACK(PROCBASE+2);    END;
           MOVE.CARD(BASE4+1386,BASE4+1397);
        END ELSE MOVE.CARD(BASE4+1397,BASE4+1397);
        GO TO L275;
P29:    PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+29) THEN BEGIN
           CHECK.STACK(14);
           CHECK.STACK(17);
           CHECK.STACK(19);
           CHECK.STACK(24);
           CHECK.STACK(ARRAYBASE+12);
           MOVE.CARD(BASE4+1437,BASE4+1526);
        END ELSE MOVE.CARD(BASE4+1525,BASE4+1526);
        GO TO L275;
P30:    PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+30) THEN BEGIN
           CHECK.STACK(14);
           CHECK.STACK(ARRAYBASE+12);
           MOVE.CARD(BASE4+1530,BASE4+1543);
        END ELSE MOVE.CARD(BASE4+1543,BASE4+1543);
        GO TO L275;
P31:    PROCESS.VARIABLES.EXCEPT (ARRAYBASE+12);
        IF ¬IN.THE.STACK(PROCBASE+31) THEN BEGIN
           IF ¬IN.THE.STACK(PROCBASE+5) THEN BEGIN
              CHECK.STACK(18);
              CHECK.STACK(35);
```

```
            CHECK.STACK(36);
            LOAD.STACK(SPECBASE+16);
            LOAD.STACK(SPECBASE+40);
            LOAD.STACK(SPECBASE+41);
            LOAD.STACK(SPECBASE+52);
            LOAD.STACK(SPECBASE+51);
            IF ¬IN.THE.STACK(PROCBASE+30) THEN BEGIN
               MOVE.CARD(BASE3+117,BASE3+119);
               MOVE.CARD(BASE4+1530,BASE4+1542);
               LOAD.STACK(PROCBASE+30);   END;
            IF ¬IN.THE.STACK(PROCBASE+45) THEN BEGIN
               MOVE.CARD(BASE4+2194,BASE4+2204);
               LOAD.STACK(PROCBASE+45);   END;
            IF ¬ IN.THE.STACK(PROCBASE+34) THEN BEGIN
               MOVE.CARD(BASE4+1725,BASE4+1757);
               LOAD.STACK(PROCBASE+34);   END;
            MOVE.CARD(BASE3+10,BASE3+16);
            MOVE.CARD(BASE4+141,BASE4+146);
            MOVE.CARD(BASE2+36,BASE2+37);
            MOVE.CARD(BASE4+147,BASE4+150);
            MOVE.CARD(BASE2+36,BASE2+37);
            MOVE.CARD(BASE4+151,BASE4+153);   END;
         LOAD.STACK(SPECBASE+15);
         MOVE.CARD(BASE3+120,BASE3+122);
         MOVE.CARD(BASE4+1549,BASE4+1581);
      END ELSE MOVE.CARD(BASE4+1581,BASE4+1581);
      GO TO L27>;
P33:  PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+33) THEN BEGIN
         CHECK.STACK(ARRAYBASE+32);
         CHECK.STACK(ARRAYBASE+28);
         CHECK.STACK(ARRAYBASE+13);
         LOAD.STACK(39);
         LOAD.STACK(SPECBASE+29);
         MOVE.CARD(BASE3+123,BASE3+126);
         MOVE.CARD(BASE4+1679,BASE4+1719);
      END ELSE MOVE.CARD(BASE4+1719,BASE4+1719);
      GO TO L27>;
P34:  GO TO L27>;
P41:  PROCESS.VARIABLES.UNTIL (ARRAYBASE+17);
      IF ¬IN.THE.STACK(PROCBASE+41) THEN BEGIN
         MOVE.CARD(BASE3+142,BASE3+144);
         LOAD.STACK(ARRAYBASE+17);   END;
         MOVE.CARD(BASE3+145,BASE3+147);
      PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+41) THEN BEGIN
         CHECK.STACK(11);
         CHECK.STACK(ARRAYBASE+7);
         CHECK.STACK(ARRAYBASE+17);
         LOAD.STACK(SPECBASE+50);
         LOAD.STACK(SPECBASE+43);
         LOAD.STACK(SPECBASE+42);
         MOVE.CARD(BASE4+1981,BASE4+2012);
      END ELSE MOVE.CARD(BASE4+2012,BASE4+2012);
      GO TO L27>;
      END;
   PHASE.TWO.ASSEMBLE;
   END;
   BEGIN
   PRINT(<'THE SEGMENT IS STORED',E>);
   END;




   BEGIN
SY SEGMENT 55
SY RELEASE WHAT
SY RELEASE SYMBOLIC LIBRARY
AL COMMENT   THE HYDRO COMPILER - PHASE TWO,SECTION THREE;
      OWN INTEGER I,J,K, ARRAYBASE.PROCBASE,SPECBASE,STORBASE,
      STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
      PAIRS,TEMP1,TEMP2;
      OWN LOGIC TYPEDATA;
      OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
      OWN INTEGER ARRAY STACK[1:1000];
      OWN INTEGER ARRAY PROGRAM[1:500,1:2];
   PROCEDURE PHASE.TWO.ASSEMBLE;
      BEGIN
      INTEGER BASE1,BASE2,BASE3,BASE4;
      BOOLEAN DECLARED;
      LIBRARY PROCEDURE LINK;
      PROCEDURE MOVE,CARD (A,B);
         VALUE A,B;
         INTEGER A,B;
         BEGIN
         PAIRS ← PAIRS+1;
         PROGRAM[PAIRS,1] ← A;
         PROGRAM[PAIRS,2] ← B;
         END;
      BOOLEAN PROCEDURE IN.THE.STACK (A);
         VALUE A;
         INTEGER A;
         BEGIN
         IN.THE.STACK ← FALSE;
         FOR I ← OPNO-1 STEP -1 UNTIL STORBASE DO
            IF STACK[I] = A THEN BEGIN
               IN.THE.STACK ← TRUE;
               GO TO L200;   END;
         L200:   END;
      PROCEDURE CHECK.STACK (A);
         VALUE A;
         INTEGER A;
         BEGIN
         IF ¬IN.THE.STACK(A) THEN BEGIN
            ERROR ← TRUE;
            PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
            'A PROCEDURE HAS BEEN CALLED WITHOUT ALL ',
            'OF THE DATA REQUIRED FOR IT',E,'BEING ENTERED.  ',
      'THIS DATA CAN BE ENTERED ON DATA CARDS PLACED',E,
      'IMMEDIATELY FOLLOWING THE PROCEDURE CALL, OR BY ',
      'MEANS OF CALLS',E,'ON OTHER PROCEDURES WHICH ',
      'CALCULATE THE DATA.  YOU MAY ALSO HAVE',E,
      'NEGLECTED TO CALL A PROCEDURE WHICH IS REQUIRED ',
      'IN YOUR PROGRAM.',E>);
      END;
   END;
PROCEDURE LOAD.STACK (A);
   VALUE A;
   INTEGER A;
   BEGIN
   STORBASE ← STORBASE-1;
   STACK[STORBASE] ← A;
   END;
PROCEDURE SINGLE.VARIABLE;
   BEGIN
   J ← STACK[OPNO]+BASE1+1;
   MOVE.CARD(J,J);
   END;
PROCEDURE ARRAY.VARIABLE;
   BEGIN
   PROCEDURE TEST.BOUNDS (A,B,C);
      VALUE A,B,C;
      INTEGER A,B,C;
      BEGIN
      IF IN.THE.STACK(A) ∧ IN.THE.STACK(B) ∧ IN.THE.STACK(C)
      THEN GO TO L240
      ELSE ERROR ← TRUE;
      PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
      'DATA ARE GIVEN FOR AN ARRAY BEFORE ONE OF THE ',
      'BOUNDS ON THAT ARRAY',E,'IS DEFINED.  ',
      'THE BOUND CAN BE DEFINED BY A DATA READ ABOVE ',
      'THAT OF',E,'THE ARRAY, OR BY A CALL ON A PROCEDURE ',
      'WHICH DEFINES IT.',E>);
      GO TO L250;
      END;
    SWITCH ARRY ← A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
      A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,
      A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,
      A31,A32,A33,A34,A35,A36,A37,A38,A39,A40,
      A41,A42,A43,A44,A45,A46,A47,A48,A49,A50,
      A51,A52,A53,A54,A55,A56;
    IF IN.THE.STACK(STACK[OPNO]) THEN DECLARED ← TRUE
      ELSE DECLARED ← FALSE;
    J ← STACK[OPNO] - ARRAYBASE;
    K ← BASE2+2+(J-1)×3;
    IF ¬DECLARED THEN GO TO ARRY[J]
    ELSE GO TO L245;
      A1:    TEST.BOUNDS(43,0,0);
      A2:    TEST.BOUNDS(40,0,0);
      A3:    TEST.BOUNDS(29,0,0);
      A4:    TEST.BOUNDS(46,0,0);
      A5:    TEST.BOUNDS(45,0,0);
      A6:    TEST.BOUNDS(49,44,0);
```

```
            A7:    TEST.WOUNDS(12,11,0);
            A8:    TEST.WOUNDS(30,0,0);
            A9:    TEST.WOUNDS(40,0,0);
            A10:   TEST.WOUNDS(41,0,0);
            A11:   TEST.WOUNDS(42,0,0);
            A12:   TEST.WOUNDS(35,0,0);
            A13:   TEST.WOUNDS(38,37,0);
            A14:   TEST.WOUNDS(40,0,0);
            A15:   TEST.WOUNDS(41,0,0);
            A16:   TEST.WOUNDS(42,0,0);
            A17:   TEST.WOUNDS(12,11,0);
            A18:   TEST.WOUNDS(48,0,0);
            A19:   TEST.WOUNDS(45,0,0);
            A20:   TEST.WOUNDS(33,0,0);
            A21:   TEST.WOUNDS(33,0,0);
            A22:   TEST.WOUNDS(33,0,0);
            A23:   TEST.WOUNDS(43,0,0);
            A24:   TEST.WOUNDS(45,0,0);
            A25:   TEST.WOUNDS(30,0,0);
            A26:   TEST.WOUNDS(34,0,0);
            A27:   TEST.WOUNDS(32,0,0);
            A28:   TEST.WOUNDS(38,0,0);
            A29:   TEST.WOUNDS(41,0,0);
            A30:   TEST.WOUNDS(44,0,0);
            A31:   TEST.WOUNDS(44,0,0);
            A32:   TEST.WOUNDS(38,0,0);
            A33:   TEST.WOUNDS(41,0,0);
            A34:   TEST.WOUNDS(42,0,0);
            A35:   TEST.WOUNDS(50,53,54);
            A36:   TEST.WOUNDS(60,0,0);
            A37:   TEST.WOUNDS(50,53,54);
            A38:   TEST.WOUNDS(50,53,54);
            A39:   TEST.WOUNDS(50,53,54);
            A40:   TEST.WOUNDS(50,53,54);
            A41:   TEST.WOUNDS(61,0,0);
            A42:   TEST.WOUNDS(50,53,0);
            A43:   TEST.WOUNDS(50,53,0);
            A44:   TEST.WOUNDS(50,53,54);
            A45:   TEST.WOUNDS(50,53,55);
            A46:   TEST.WOUNDS(50,53,0);
            A47:   TEST.WOUNDS(50,53,0);
            A48:   TEST.WOUNDS(50,53,0);
            A49:   TEST.WOUNDS(50,53,0);
            A50:   TEST.WOUNDS(50,53,0);
            A51:   TEST.WOUNDS(61,0,0);
            A52:   TEST.WOUNDS(50,53,54);
            A53:   TEST.WOUNDS(50,53,55);
            A54:   TEST.WOUNDS(50,53,55);
            A55:   TEST.WOUNDS(50,53,54);
            A56:   TEST.WOUNDS(50,53,0);
    L240:   MOVE.CARD(K,K+2);   GO TO L250;
    L245:   MOVE.CARD(K+1,K+2);
    L250:   END;
PROCEDURE PROCESS.VARIABLES;
    BEGIN
```

```
      L206:   OPNO ← OPNO+1;
      IF STACK[OPNO] < ARRAYBASE THEN BEGIN
         SINGLE.VARIABLE;
         GO TO L206;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
         ARRAY.VARIABLE;
         GO TO L206;   END;
      END;
   PROCEDURE PROCESS.VARIABLES.EXCEPT (A);
      VALUE A;
      INTEGER A;
      BEGIN
      L207:   OPNO ← OPNO+1;
      IF STACK[OPNO] = A THEN GO TO L208;
      IF STACK[OPNO] < ARRAYBASE THEN BEGIN
         SINGLE.VARIABLE;
         GO TO L207;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
         ARRAY.VARIABLE;
         GO TO L207;   END;
      L208:   OPNO ← OPNO+1;
         IF STACK[OPNO] < PROCBASE THEN GO TO L208;
      END;
   PROCEDURE PROCESS.VARIABLES.UNTIL (A);
      VALUE A;
      INTEGER A;
      BEGIN
      L209:   OPNO ← OPNO+1;
      IF STACK[OPNO] = A THEN GO TO L210;
      IF STACK[OPNO] < ARRAYBASE THEN BEGIN
         SINGLE.VARIABLE;
         GO TO L209;   END;
      IF STACK[OPNO] < PROCBASE THEN BEGIN
         ARRAY.VARIABLE;
         GO TO L209;   END;
      L210:   OPNO ← OPNO-1;
      END;
COMMENT   CODE FOR THE PHASE.TWO PROCEDURES;
   SWITCH PROC ← P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,
         P11,P12,P13,P14,P15,P16,P17,P18,P19,P20,
         P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,
         P31,P32,P33,P34,P35,P36,P37,P38,P39,P40,
         P41,P42,P43,P44,P45,P46,P47,P48,P49,P50,
         P51,P52,P53,P54,P55,P56,P57,P58,P59,P60,
         P61,P62,P63,P64,P65,P66,P67,P68,P69,P70,
         P71,P72,P73,P74,P75,P76,P77,P78,P79,P80,
         P81,P82,P83,P84,P85;
   SWITCH BACK ← L46,L47,L48,L49,L50,L51,L52,
                 L53,L54,L55,L56,L57,L58,L59;
   BASE1 ← 262;
   BASE2 ← 327;
   BASE3 ← 497;
   BASE4 ← 696;
   L275: STACK[TEMP1] ← TEMP2;
         J ← STACK[OPNO] - PROCBASE + 1;
```

Reproduced from
best available copy.

```
        TEMP1 ← OPNO;
        TEMP2 ← STACK[OPNO];
        STACK[OPNO] ← 0;
        GO TO PROC[J];
  SAME: IF ¬IN.THE.STACK(PROCBASE+77) THEN BEGIN
            MOVE.CARD(BASE3+156,BASE3+157);
            IF ¬IN.THE.STACK(ARRAYBASE+35) THEN BEGIN
               MOVE.CARD(BASE3+158,BASE3+158);
               LOAD.STACK(ARRAYBASE+35);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+37) THEN BEGIN
               MOVE.CARD(BASE3+159,BASE3+159);
               LOAD.STACK(ARRAYBASE+37);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+38) THEN BEGIN
               MOVE.CARD(BASE3+160,BASE3+160);
               LOAD.STACK(ARRAYBASE+38);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+39) THEN BEGIN
               MOVE.CARD(BASE3+161,BASE3+161);
               LOAD.STACK(ARRAYBASE+39);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+40) THEN BEGIN
               MOVE.CARD(BASE3+162,BASE3+162);
               LOAD.STACK(ARRAYBASE+40);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+44) THEN BEGIN
               MOVE.CARD(BASE3+163,BASE3+163);
               LOAD.STACK(ARRAYBASE+44);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+52) THEN BEGIN
               MOVE.CARD(BASE3+164,BASE3+164);
               LOAD.STACK(ARRAYBASE+52);    END;
            IF ¬IN.THE.STACK(SPECBASE+66) THEN BEGIN
               MOVE.CARD(BASE3+165,BASE3+165);
               LOAD.STACK(SPECBASE+66);   END;
            IF ¬IN.THE.STACK(SPECBASE+68) THEN BEGIN
               MOVE.CARD(BASE3+166,BASE3+166);
               LOAD.STACK(SPECBASE+68);   END;
            IF ¬IN.THE.STACK(SPECBASE+70) THEN BEGIN
               MOVE.CARD(BASE3+167,BASE3+167);
               LOAD.STACK(SPECBASE+70);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+45) THEN BEGIN
               MOVE.CARD(BASE3+168,BASE3+168);
               LOAD.STACK(ARRAYBASE+45);    END;
        MOVE.CARD(BASE4+3436,BASE4+3549);
        LOAD.STACK(PROCBASE+77); END;
        GO TO BACK[J];
  P0:   IF ¬ ERROR THEN BEGIN
            MOVE.CARD(1,6);
            LINK(57);
            END ELSE HALT;
  P2: P3: P6: P7: P8: P9: P10: P11: P12: P13: P14: P15: P16:
  P21: P22: P26: P28: P32: P35: P36: P37: P38: P39: P40:
  P42: P43: P44: P45: P81: P82: P83: P84: P85:      LINK(53);
  P1: P4: P5: P17: P18: P19: P20: P23: P24: P25: P27: P29:
  P30: P31: P33: P34: P41:          LINK(54);
  P60: P61: P62: P63: P64: P65: P66: P67: P68: P69:
  P70: P71: P72: P73: P74: P75: P76: P77: P78: P79: P80:  LINK(56);
  P46:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+46) THEN BEGIN
```

```
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+52);
             J ← 1;   GO TO SAME;
             L46: MOVE.CARD(BASE4+2212,BASE4+2230);
          LOAD.STACK(SPECBASE+68);
          END ELSE MOVE.CARD(BASE4+2230,BASE4+2230);
          GO TO L275;
  P47: PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+47) THEN BEGIN
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+52);
             J ← 2;   GO TO SAME;
             L47: MOVE.CARD(BASE4+2236,BASE4+2254);
          LOAD.STACK(ARRAYBASE+40);
          END ELSE MOVE.CARD(BASE4+2254,BASE4+2254);
          GO TO L275;
  P48: PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+48) THEN BEGIN
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+52);
             J ← 3;   GO TO SAME;
             L48: MOVE.CARD(BASE4+2260,BASE4+2278);
          LOAD.STACK(SPECBASE +66);
          END ELSE MOVE.CARD(BASE4+2278,BASE4+2278);
          GO TO L275;
  P49: PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+49) THEN BEGIN
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+45);
          CHECK.STACK(ARRAYBASE+52);
             J ← 4;   GO TO SAME;
             L49: MOVE.CARD(BASE4+2284,BASE4+2302);
          LOAD.STACK(ARRAYBASE+35);
          END ELSE MOVE.CARD(BASE4+2302,BASE4+2302);
          GO TO L275;
  P50: PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+50) THEN BEGIN
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+45);
          CHECK.STACK(ARRAYBASE+52);
```

```
         J ← 5;    GO TO SAME;
         L50; MOVE.CARD(BASE4+2308,BASE4+2326);
      LOAD.STACK(SPECBASE +70);
      END ELSE MOVE.CARD(BASE4+2326,BASE4+2326);
      GO TO L275;
P51; PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+51) THEN BEGIN
      CHECK.STACK(ARRAYBASE+47);
      CHECK.STACK(ARRAYBASE+46);
      CHECK.STACK(ARRAYBASE+53);
      CHECK.STACK(ARRAYBASE+54);
      CHECK.STACK(ARRAYBASE+45);
      CHECK.STACK(ARRAYBASE+52);
         J ← 6;    GO TO SAME;
         L51; MOVE.CARD(BASE4+2332,BASE4+2350);
      LOAD.STACK(ARRAYBASE+44);
      END ELSE MOVE.CARD(BASE4+2350,BASE4+2350);
      GO TO L275;
P52; PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+52) THEN BEGIN
      CHECK.STACK(ARRAYBASE+47);
      CHECK.STACK(ARRAYBASE+46);
      CHECK.STACK(ARRAYBASE+53);
      CHECK.STACK(ARRAYBASE+54);
      CHECK.STACK(ARRAYBASE+52);
         J ← 7;    GO TO SAME;
         L52; MOVE.CARD(BASE4+2356,BASE4+2375);
      LOAD.STACK(ARRAYBASE+39);
      END ELSE MOVE.CARD(BASE4+2375,BASE4+2375);
      GO TO L275;
P53; PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+53) THEN BEGIN
      CHECK.STACK(ARRAYBASE+47);
      CHECK.STACK(ARRAYBASE+46);
      CHECK.STACK(ARRAYBASE+53);
      CHECK.STACK(ARRAYBASE+54);
      CHECK.STACK(ARRAYBASE+45);
      CHECK.STACK(ARRAYBASE+52);
         J ← 8;    GO TO SAME;
         L53; MOVE.CARD(BASE4+2381,BASE4+2399);
      LOAD.STACK(ARRAYBASE+37);
      END ELSE MOVE.CARD(BASE4+2399,BASE4+2399);
      GO TO L275;
P54; PROCESS.VARIABLES;
      IF ¬IN.THE.STACK(PROCBASE+54) THEN BEGIN
      CHECK.STACK(ARRAYBASE+47);
      CHECK.STACK(ARRAYBASE+46);
      CHECK.STACK(ARRAYBASE+53);
      CHECK.STACK(ARRAYBASE+54);
      CHECK.STACK(ARRAYBASE+45);
      CHECK.STACK(ARRAYBASE+52);
         J ← 9;    GO TO SAME;
         L54; MOVE.CARD(BASE4+2405,BASE4+2424);
      LOAD.STACK(ARRAYBASE+38);
      END ELSE MOVE.CARD(BASE4+2424,BASE4+2424);
```

```
        GO TO L275;
P55:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+55) THEN BEGIN
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+52);
           J ← 10;   GO TO SAME;
           L55: MOVE.CARD(BASE4+2430,BASE4+2458);
        LOAD.STACK(ARRAYBASE+40);
        LOAD.STACK(SPECBASE +66);
        LOAD.STACK(ARRAYBASE+35);
        LOAD.STACK(SPECBASE +70);
        LOAD.STACK(ARRAYBASE+44);
        LOAD.STACK(ARRAYBASE+39);
        LOAD.STACK(ARRAYBASE+37);
        LOAD.STACK(ARRAYBASE+38);
        LOAD.STACK(SPECBASE+68);
        END ELSE MOVE.CARD(BASE4+2458,BASE4+2458);
        GO TO L275;
P56:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+56) THEN BEGIN
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+52);
           J ← 11;   GO TO SAME;
           L56: MOVE.CARD(BASE4+2471,BASE4+2491);
        LOAD.STACK(ARRAYBASE+40);
        LOAD.STACK(ARRAYBASE+35);
        END ELSE MOVE.CARD(BASE4+2491,BASE4+2491);
        GO TO L275;
P57:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+57) THEN BEGIN
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+52);
           J ← 12;   GO TO SAME;
           L57: MOVE.CARD(BASE4+2497,BASE4+2516);
        LOAD.STACK(ARRAYBASE+40);
        LOAD.STACK(ARRAYBASE+38);
        END ELSE MOVE.CARD(BASE4+2516,BASE4+2516);
        GO TO L275;
P58:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+58) THEN BEGIN
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+53);
```

```
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+52);
             J ← 13;  GO TO SAME;
             L58: MOVE.CARD(BASE4+2522,BASE4+2542);
          LOAD.STACK(ARRAYBASE+40);
          LOAD.STACK(ARRAYBASE+39);
          END ELSE MOVE.CARD(BASE4+2542,BASE4+2542);
          GO TO L275;
     P59: PROCESS.VARIABLES;
          IF ¬IN.THE.STACK(PROCBASE+59) THEN BEGIN
          CHECK.STACK(ARRAYBASE+47);
          CHECK.STACK(ARRAYBASE+46);
          CHECK.STACK(ARRAYBASE+53);
          CHECK.STACK(ARRAYBASE+54);
          CHECK.STACK(ARRAYBASE+45);
          CHECK.STACK(ARRAYBASE+52);
             J ← 14;  GO TO SAME;
             L59: MOVE.CARD(BASE4+2548,BASE4+2570);
          LOAD.STACK(ARRAYBASE+40);
          LOAD.STACK(ARRAYBASE+44);
          LOAD.STACK(ARRAYBASE+37);
          END ELSE MOVE.CARD(BASE4+2570,BASE4+2570);
          GO TO L275;
     END;
  PHASE.TWO.ASSEMBLE;
  END;
  BEGIN
  PRINT(<'THE SEGMENT IS STORED',E>);
  END;



  BEGIN
SY SEGMENT 56
SY RELEASE WHAT
SY RELEASE SYMBOLIC LIBRARY
AL COMMENT   THE HYDRO COMPILER - PHASE TWO,SECTION FOUR;
     OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
     STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
     PAIRS,TEMP1,TEMP2;
     OWN LOGIC TYPEDATA;
     OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
     OWN INTEGER ARRAY STACK[1:1000];
     OWN INTEGER ARRAY PROGRAM[1:500,1:2];
  PROCEDURE PHASE.TWO.ASSEMBLE;
     BEGIN
     INTEGER BASE1,BASE2,BASE3,BASE4;
     BOOLEAN DECLARED;
     LIBRARY PROCEDURE LINK;
     PROCEDURE MOVE.CARD (A,B);
        VALUE A,B;
        INTEGER A,B;
        BEGIN
```

```
      PAIRS ← PAIRS+1;
      PROGRAM[PAIRS,1] ← A;
      PROGRAM[PAIRS,2] ← B;
      END;
   BOOLEAN PROCEDURE IN.THE.STACK (A);
      VALUE A;
      INTEGER A;
      BEGIN
      IN.THE.STACK ← FALSE;
      FOR I ← OPNO-1 STEP -1 UNTIL STORBASE DO
         IF STACK[I] = A THEN BEGIN
             IN.THE.STACK ← TRUE;
             GO TO L200;   END;
      L200:   END;
   PROCEDURE CHECK.STACK (A);
      VALUE A;
      INTEGER A;
      BEGIN
      IF ¬IN.THE.STACK(A) THEN BEGIN
         ERROR ← TRUE;
         PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
         'A PROCEDURE HAS BEEN CALLED WITHOUT ALL ',
         'OF THE DATA REQUIRED FOR IT',E,'BEING ENTERED.  ',
         'THIS DATA CAN BE ENTERED ON DATA CARDS PLACED',E,
         'IMMEDIATELY FOLLOWING THE PROCEDURE CALL, OR BY ',
         'MEANS OF CALLS',E,'ON OTHER PROCEDURES WHICH ',
         'CALCULATE THE DATA.  YOU MAY ALSO HAVE',E,
         'NEGLECTED TO CALL A PROCEDURE WHICH IS REQUIRED ',
         'IN YOUR PROGRAM.',E>);
         END;
      END;
   PROCEDURE LOAD.STACK (A);
      VALUE A;
      INTEGER A;
      BEGIN
      STORBASE ← STORBASE-1;
      STACK[STORBASE] ← A;
      END;
   PROCEDURE SINGLE.VARIABLE;
      BEGIN
      J ← STACK[OPNO]+BASE1+1;
      MOVE.CARD(J,J);
      END;
   PROCEDURE ARRAY.VARIABLE;
      BEGIN
      PROCEDURE TEST.BOUNDS (A,B,C);
         VALUE A,B,C;
         INTEGER A,B,C;
         BEGIN
         IF IN.THE.STACK(A) ∧ IN.THE.STACK(B) ∧ IN.THE.STACK(C)
         THEN GO TO L240
         ELSE ERROR ← TRUE;
         PRINT(<120B,1C,'YOUR PROGRAM IS IN ERROR',E,
         'DATA ARE GIVEN FOR AN ARRAY BEFORE ONE OF THE ',
         'BOUNDS ON THAT ARRAY',E,'IS DEFINED.  ',
         'THE BOUND CAN BE DEFINED BY A DATA READ ABOVE ',
         'THAT OF',E,'THE ARRAY, OR BY A CALL ON A PROCEDURE ',
         'WHICH DEFINES IT.',E>);
         GO TO L250;
         END;
     SWITCH ARRY ← A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
         A11,A12,A13,A14,A15,A16,A17,A18,A19,A20,
         A21,A22,A23,A24,A25,A26,A27,A28,A29,A30,
         A31,A32,A33,A34,A35,A36,A37,A38,A39,A40,
         A41,A42,A43,A44,A45,A46,A47,A48,A49,A50,
         A51,A52,A53,A54,A55,A56;
     IF IN.THE.STACK(STACK(OPNO)) THEN DECLARED ← TRUE
         ELSE DECLARED ← FALSE;
     J ← STACK(OPNO) - ARRAYBASE;
     K ← BASE2+2+(J-1)*3;
     IF ¬DECLARED THEN GO TO ARRY(J)
     ELSE GO TO L245;
         A1:    TEST.BOUNDS(43,0,0);
         A2:    TEST.BOUNDS(40,0,0);
         A3:    TEST.BOUNDS(29,0,0);
         A4:    TEST.BOUNDS(46,0,0);
         A5:    TEST.BOUNDS(45,0,0);
         A6:    TEST.BOUNDS(49,44,0);
         A7:    TEST.BOUNDS(12,11,0);
         A8:    TEST.BOUNDS(30,0,0);
         A9:    TEST.BOUNDS(40,0,0);
         A10:   TEST.BOUNDS(41,0,0);
         A11:   TEST.BOUNDS(42,0,0);
         A12:   TEST.BOUNDS(35,0,0);
         A13:   TEST.BOUNDS(38,37,0);
         A14:   TEST.BOUNDS(40,0,0);
         A15:   TEST.BOUNDS(41,0,0);
         A16:   TEST.BOUNDS(42,0,0);
         A17:   TEST.BOUNDS(12,11,0);
         A18:   TEST.BOUNDS(48,0,0);
         A19:   TEST.BOUNDS(45,0,0);
         A20:   TEST.BOUNDS(33,0,0);
         A21:   TEST.BOUNDS(33,0,0);
         A22:   TEST.BOUNDS(33,0,0);
         A23:   TEST.BOUNDS(43,0,0);
         A24:   TEST.BOUNDS(45,0,0);
         A25:   TEST.BOUNDS(30,0,0);
         A26:   TEST.BOUNDS(34,0,0);
         A27:   TEST.BOUNDS(32,0,0);
         A28:   TEST.BOUNDS(38,0,0);
         A29:   TEST.BOUNDS(41,0,0);
         A30:   TEST.BOUNDS(44,0,0);
         A31:   TEST.BOUNDS(44,0,0);
         A32:   TEST.BOUNDS(38,0,0);
         A33:   TEST.BOUNDS(41,0,0);
         A34:   TEST.BOUNDS(42,0,0);
         A35:   TEST.BOUNDS(50,53,54);
         A36:   TEST.BOUNDS(60,0,0);
         A37:   TEST.BOUNDS(50,53,54);
         A38:   TEST.BOUNDS(50,53,54);
```

```
      A39:  TEST.BOUNDS(50,53,54);
      A40:  TEST.BOUNDS(50,53,54);
      A41:  TEST.BOUNDS(61,0,0);
      A42:  TEST.BOUNDS(50,53,0);
      A43:  TEST.BOUNDS(50,53,0);
      A44:  TEST.BOUNDS(50,53,54);
      A45:  TEST.BOUNDS(50,53,55);
      A46:  TEST.BOUNDS(50,53,0);
      A47:  TEST.BOUNDS(50,53,0);
      A48:  TEST.BOUNDS(50,53,0);
      A49:  TEST.BOUNDS(50,53,0);
      A50:  TEST.BOUNDS(50,53,0);
      A51:  TEST.BOUNDS(61,0,0);
      A52:  TEST.BOUNDS(50,53,54);
      A53:  TEST.BOUNDS(50,53,55);
      A54:  TEST.BOUNDS(50,53,55);
      A55:  TEST.BOUNDS(50,53,54);
      A56:  TEST.BOUNDS(50,53,0);
   L240:   MOVE.CARD(K,K+2);    GO TO L250;
   L245:   MOVE.CARD(K+1,K+2);
   L250:   END;
PROCEDURE PROCESS.VARIABLES;
   BEGIN
   L206:   OPNO ← OPNO+1;
   IF STACK[OPNO] < ARRAYBASE THEN BEGIN
      SINGLE.VARIABLE;
      GO TO L206;   END;
   IF STACK[OPNO] < PROCBASE THEN BEGIN
      ARRAY.VARIABLE;
      GO TO L206;   END;
   END;
PROCEDURE PROCESS.VARIABLES.EXCEPT (A);
   VALUE A;
   INTEGER A;
   BEGIN
   L207:   OPNO ← OPNO+1;
   IF STACK[OPNO] = A THEN GO TO L208;
   IF STACK[OPNO] < ARRAYBASE THEN BEGIN
      SINGLE.VARIABLE;
      GO TO L207;   END;
   IF STACK[OPNO] < PROCBASE THEN BEGIN
      ARRAY.VARIABLE;
      GO TO L207;   END;
   L208:   OPNO ← OPNO+1;
      IF STACK[OPNO] < PROCBASE THEN GO TO L208;
   END;
PROCEDURE PROCESS.VARIABLES.UNTIL (A);
   VALUE A;
   INTEGER A;
   BEGIN
   L209:   OPNO ← OPNO+1;
   IF STACK[OPNO] = A THEN GO TO L210;
   IF STACK[OPNO] < ARRAYBASE THEN BEGIN
      SINGLE.VARIABLE;
      GO TO L209;   END;
```

```
        IF STACK(OPNO) < PROCBASE THEN BEGIN
           ARRAY.VARIABLE;
           GO TO L209;   END;
        L210:   OPNO ← OPNO-1;
        END;
COMMENT   CODE FOR THE PHASE,TWO PROCEDURE;
   SWITCH PROC ← P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,
          P11,P12,P13,P14,P15,P16,P17,P18,P19,P20,
          P21,P22,P23,P24,P25,P26,P27,P28,P29,P30,
          P31,P32,P33,P34,P35,P36,P37,P38,P39,P40,
          P41,P42,P43,P44,P45,P46,P47,P48,P49,P50,
          P51,P52,P53,P54,P55,P56,P57,P58,P59,P60,
          P61,P62,P63,P64,P65,P66,P67,P68,P69,P70,
          P71,P72,P73,P74,P75,P76,P77,P78,P79,P80,
          P81,P82,P83,P84,P85;
   SWITCH BACK ← L71,L72,L73,L74,L75;
   BASE1 ← 262;
   BASE2 ← 327;
   BASE3 ← 497;
   BASE4 ← 696;
   L275: STACK(TEMP1) ← TEMP2;
         J ← STACK(OPNO) - PROCBASE + 1;
         TEMP1 ← OPNO;
         TEMP2 ← STACK(OPNO);
         STACK(OPNO) ← 0;
         GO TO PROC(J);
   SAME: IF ¬IN.THE.STACK(PROCBASE+77) THEN BEGIN
            MOVE.CARD(BASE3+156,BASE3+157);
            IF ¬IN.THE.STACK(ARRAYBASE+35) THEN BEGIN
               MOVE.CARD(BASE3+158,BASE3+158);
               LOAD.STACK(ARRAYBASE+35);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+37) THEN BEGIN
               MOVE.CARD(BASE3+159,BASE3+159);
               LOAD.STACK(ARRAYBASE+37);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+38) THEN BEGIN
               MOVE.CARD(BASE3+160,BASE3+160);
               LOAD.STACK(ARRAYBASE+38);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+39) THEN BEGIN
               MOVE.CARD(BASE3+161,BASE3+161);
               LOAD.STACK(ARRAYBASE+39);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+40) THEN BEGIN
               MOVE.CARD(BASE3+162,BASE3+162);
               LOAD.STACK(ARRAYBASE+40);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+44) THEN BEGIN
               MOVE.CARD(BASE3+163,BASE3+163);
               LOAD.STACK(ARRAYBASE+44);   END;
            IF ¬IN.THE.STACK(ARRAYBASE+52) THEN BEGIN
               MOVE.CARD(BASE3+164,BASE3+164);
               LOAD.STACK(ARRAYBASE+52);   END;
            IF ¬IN.THE.STACK(SPECBASE+66) THEN BEGIN
               MOVE.CARD(BASE3+165,BASE3+165);
               LOAD.STACK(SPECBASE+66);   END;
            IF ¬IN.THE.STACK(SPECBASE+68) THEN BEGIN
               MOVE.CARD(BASE3+166,BASE3+166);
               LOAD.STACK(SPECBASE+68);   END;
            IF ¬IN.THE.STACK(SPECBASE+70) THEN BEGIN
               MOVE.CARD(BASE3+167,BASE3+167);
               LOAD.STACK(SPECBASE+70);    END;
            IF ¬IN.THE.STACK(ARRAYBASE+45) THEN BEGIN
               MOVE.CARD(BASE3+168,BASE3+168);
               LOAD.STACK(ARRAYBASE+45);    END;
         MOVE.CARD(BASE4+3436,BASE4+3549);
         LOAD.STACK(PROCBASE+77); END;
         GO TO BACK(J);
P0:      IF ¬ ERROR THEN BEGIN
            MOVE.CARD(1,6);
            LINK(57);
            END ELSE HALT;
P2: P3: P6: P7: P8: P9: P10: P11: P12: P13: P14: P15: P16:
P21: P22: P26: P28: P32: P35: P36: P37: P38: P39: P40:
P42: P43: P44: P45: P81: P82: P83: P84: P85:      LINK(53);
P1: P4: P5: P17: P18: P19: P20: P23: P24: P25: P27: P29:
P30: P31: P33: P34: P41:             LINK(54);
P46: P47: P48: P49: P50: P51: P52: P53: P54: P55:
P56: P57: P58: P59:                  LINK(55);
P60: PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+60) THEN BEGIN
         CHECK.STACK(ARRAYBASE+42);
         LOAD.STACK(ARRAYBASE+48);
         MOVE.CARD(BASE3+169,BASE3+170);
         MOVE.CARD(BASE4+2576,BASE4+2590);
         END ELSE MOVE.CARD(BASE4+2590,BASE4+2590);
         GO TO L275;
P61: PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+61) THEN BEGIN
         CHECK.STACK(ARRAYBASE+43);
         CHECK.STACK(ARRAYBASE+43);
         LOAD.STACK(ARRAYBASE+49);
         MOVE.CARD(BASE3+171,BASE3+172);
         MOVE.CARD(BASE4+2596,BASE4+2612);
         END ELSE MOVE.CARD(BASE4+2612,BASE4+2612);
         GO TO L275;
P62: PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+62) THEN BEGIN
         CHECK.STACK(ARRAYBASE+46);
         CHECK.STACK(ARRAYBASE+40);
         CHECK.STACK(ARRAYBASE+41);
         LOAD.STACK(ARRAYBASE+51);
         MOVE.CARD(BASE3+173,BASE3+174);
         MOVE.CARD(BASE4+2617,BASE4+2640);
         END ELSE MOVE.CARD(BASE4+2640,BASE4+2640);
         GO TO L275;
P63: PROCESS.VARIABLES;
         IF ¬IN.THE.STACK(PROCBASE+63) THEN BEGIN
         CHECK.STACK(ARRAYBASE+46);
         CHECK.STACK(ARRAYBASE+40);
         CHECK.STACK(ARRAYBASE+51);
         LOAD.STACK(ARRAYBASE+41);
         MOVE.CARD(BASE3+175,BASE3+175);
         MOVE.CARD(BASE4+2646,BASE4+2668);
```

```
        END ELSE MOVE.CARD(BASE4+2668,BASE4+2668);
        GO TO L275;
   P64: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+64) THEN BEGIN
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+49);
        CHECK.STACK(ARRAYBASE+40);
        CHECK.STACK(ARRAYBASE+44);
        CHECK.STACK(ARRAYBASE+37);
        LOAD.STACK(ARRAYBASE+41);
        MOVE.CARD(BASE3+177,BASE3+179);
        MOVE.CARD(BASE4+2674,BASE4+2697);
        END ELSE MOVE.CARD(BASE4+2697,BASE4+2697);
        GO TO L275;
   P65: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+65) THEN BEGIN
        CHECK.STACK(59);
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+40);
        CHECK.STACK(ARRAYBASE+44);
        CHECK.STACK(ARRAYBASE+37);
        CHECK.STACK(ARRAYBASE+41);
        LOAD.STACK(SPECBASE +73);
        LOAD.STACK(SPECBASE +74);
        MOVE.CARD(BASE3+180,BASE3+181);
        MOVE.CARD(BASE4+2703,BASE4+2743);
        END ELSE MOVE.CARD(BASE4+2743,BASE4+2743);
        GO TO L275;
  P66:  PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+66) THEN BEGIN
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+49);
        CHECK.STACK(ARRAYBASE+40);
        CHECK.STACK(ARRAYBASE+44);
        CHECK.STACK(ARRAYBASE+41);
        LOAD.STACK(ARRAYBASE+37);
        IF ¬IN.THE.STACK(PROCBASE+77) THEN
        MOVE.CARD(BASE3+182,BASE3+183);
        MOVE.CARD(BASE4+2748,BASE4+2775);
        END ELSE MOVE.CARD(BASE4+2775,BASE4+2775);
        GO TO L275;
   P67: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+67) THEN BEGIN
        CHECK.STACK(ARRAYBASE+46);
        CHECK.STACK(ARRAYBASE+40);
        CHECK.STACK(ARRAYBASE+35);
        CHECK.STACK(ARRAYBASE+41);
        CHECK.STACK(ARRAYBASE+55);
        LOAD.STACK(SPECBASE +65);
        MOVE.CARD(BASE3+184,BASE3+185);
        MOVE.CARD(BASE4+2781,BASE4+2805);
        END ELSE MOVE.CARD(BASE4+2805,BASE4+2805);
        GO TO L275;
   P68: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+68) THEN BEGIN
```

```
       CHECK.STACK(A2);
       CHECK.STACK(ARRAYBASE+46);
       CHECK.STACK(ARRAYBASE+40);
       CHECK.STACK(ARRAYBASE+39);
       LOAD.STACK(SPECBASE+75);
       MOVE.CARD(BASE3+186,BASE3+187);
       MOVE.CARD(BASE4+2812,BASE4+2831);
       END ELSE MOVE.CARD(BASE4+2831,BASE4+2831);
       GO TO L275;
P69:  PROCESS.VARIABLES;
       IF ¬IN.THE.STACK(PROCBASE+69) THEN BEGIN
       CHECK.STACK(ARRAYBASE+46);
       CHECK.STACK(ARRAYBASE+48);
       CHECK.STACK(ARRAYBASE+37);
       CHECK.STACK(ARRAYBASE+44);
       CHECK.STACK(ARRAYBASE+51);
       LOAD.STACK(SPECBASE +67);
       MOVE.CARD(BASE3+188,BASE3+189);
       MOVE.CARD(BASE4+2837,BASE4+2865);
       END ELSE MOVE.CARD(BASE4+2865,BASE4+2865);
       GO TO L275;
P70:  PROCESS.VARIABLES;
       IF ¬IN.THE.STACK(PROCBASE+70) THEN BEGIN
       CHECK.STACK(A2);
       CHECK.STACK(ARRAYBASE+46);
       CHECK.STACK(ARRAYBASE+40);
       CHECK.STACK(ARRAYBASE+38);
       CHECK.STACK(ARRAYBASE+41);
       LOAD.STACK(SPECBASE +77);
       MOVE.CARD(BASE3+190,BASE3+191);
       MOVE.CARD(BASE4+2872,BASE4+2898);
       END ELSE MOVE.CARD(BASE4+2898,BASE4+2898);
       GO TO L275;
P71:  PROCESS.VARIABLES;
       IF ¬IN.THE.STACK(PROCBASE+71) THEN BEGIN
       CHECK.STACK(ARRAYBASE+47);
       CHECK.STACK(ARRAYBASE+49);
       CHECK.STACK(ARRAYBASE+53);
       CHECK.STACK(ARRAYBASE+54);
       CHECK.STACK(ARRAYBASE+45);
       CHECK.STACK(ARRAYBASE+41);
       MOVE.CARD(BASE3+192,BASE3+193);
          J ← 1;   GO TO SAME;
       L71: IF ¬IN.THE.STACK(PROCBASE+76) THEN BEGIN
          MOVE.CARD(BASE4+3419,BASE4+3428);
          LOAD.STACK(PROCBASE+76);    END;
       IF ¬IN.THE.STACK(PROCBASE+78) THEN BEGIN
          MOVE.CARD(BASE4+3557,BASE4+3576);
          LOAD.STACK(PROCBASE+78);    END;
       MOVE.CARD(BASE4+2906,BASE4+2928);
       END ELSE MOVE.CARD(BASE4+2928,BASE4+2928);
       GO TO L275;
P72:  PROCESS.VARIABLES;
       IF ¬IN.THE.STACK(PROCBASE+72) THEN BEGIN
       CHECK.STACK(ARRAYBASE+47);
```

```
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+41);
        MOVE.CARD(BASE3+192,BASE3+193);
           J ← 2;    GO TO SAME;
        L72: IF ¬IN.THE.STACK(PROCBASE+76) THEN BEGIN
           MOVE.CARD(BASE4+3419,BASE4+3428);
           LOAD.STACK(PROCBASE+76);    END;
        IF ¬IN.THE.STACK(PROCBASE+78) THEN BEGIN
           MOVE.CARD(BASE4+3557,BASE4+3576);
           LOAD.STACK(PROCBASE+78);    END;
        MOVE.CARD(BASE4+2934,BASE4+2955);
        END ELSE MOVE.CARD(BASE4+2955,BASE4+2955);
        GO TO L275;
  P73: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+73) THEN BEGIN
        CHECK.STACK(ARRAYBASE+56);
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+41);
        MOVE.CARD(BASE3+192,BASE3+193);
           J ← 3;    GO TO SAME;
        L73: IF ¬IN.THE.STACK(PROCBASE+76) THEN BEGIN
           MOVE.CARD(BASE4+3419,BASE4+3428);
           LOAD.STACK(PROCBASE+76);    END;
        IF ¬IN.THE.STACK(PROCBASE+78) THEN BEGIN
           MOVE.CARD(BASE4+3557,BASE4+3576);
           LOAD.STACK(PROCBASE+78);    END;
        IF ¬IN.THE.STACK(PROCBASE+79) THEN BEGIN
           MOVE.CARD(BASE4+3584,BASE4+3613);
           LOAD.STACK(PROCBASE+79);    END;
        MOVE.CARD(BASE4+2961,BASE4+2978);
        END ELSE MOVE.CARD(BASE4+2979,BASE4+2978);
        GO TO L275;
  P74: PROCESS.VARIABLES;
        IF ¬IN.THE.STACK(PROCBASE+74) THEN BEGIN
        CHECK.STACK(56);
        CHECK.STACK(57);
        CHECK.STACK(58);
        CHECK.STACK(51);
        CHECK.STACK(ARRAYBASE+47);
        CHECK.STACK(ARRAYBASE+53);
        CHECK.STACK(ARRAYBASE+54);
        CHECK.STACK(ARRAYBASE+45);
        CHECK.STACK(ARRAYBASE+41);
        CHECK.STACK(ARRAYBASE+48);
        CHECK.STACK(ARRAYBASE+49);
        CHECK.STACK(ARRAYBASE+50);
        IF ¬IN.THE.STACK(52) ∧ ¬IN.THE.STACK(ARRAYBASE+43) THEN
        CHECK.STACK(52);
        MOVE.CARD(BASE3+192,BASE3+193);
           J ← 4;    GO TO SAME;
```

```
     L74: IF ¬IN.THE.STACK(PROCBASE+76) THEN BEGIN
        MOVE.CARD(BASE4+3419,BASE4+3428);
        LOAD.STACK(PROCBASE+76);   END;
     IF ¬IN.THE.STACK(PROCBASE+78) THEN BEGIN
        MOVE.CARD(BASE4+3557,BASE4+3576);
        LOAD.STACK(PROCBASE+78);   END;
     IF ¬IN.THE.STACK(PROCBASE+80) THEN BEGIN
        MOVE.CARD(BASE3+196,BASE3+197);
        IF ¬IN.THE.STACK(ARRAYBASE+43) THEN BEGIN
           MOVE.CARD(BASE3+198,BASE3+198);
           LOAD.STACK(ARRAYBASE+43);   END;
        MOVE.CARD(BASE4+3621,BASE4+3675);
        LOAD.STACK(PROCBASE+80);   END;
     MOVE.CARD(BASE4+2984,BASE4+3080);
     END ELSE MOVE.CARD(BASE4+3080,BASE4+3080);
     GO TO L275;
P75: PROCESS.VARIABLES;
     IF ¬IN.THE.STACK(PROCBASE+75) THEN BEGIN
     CHECK.STACK(57);
     CHECK.STACK(58);
     CHECK.STACK(ARRAYBASE+47);
     CHECK.STACK(ARRAYBASE+53);
     CHECK.STACK(ARRAYBASE+54);
     CHECK.STACK(ARRAYBASE+45);
     CHECK.STACK(ARRAYBASE+41);
     CHECK.STACK(ARRAYBASE+48);
     CHECK.STACK(ARRAYBASE+49);
     CHECK.STACK(ARRAYBASE+50);
     CHECK.STACK(ARRAYBASE+36);
     IF ¬IN.THE.STACK(52) ∧ ¬IN.THE.STACK(ARRAYBASE+43) THEN
     CHECK.STACK(52);
     MOVE.CARD(BASE3+192,BASE3+193);
     IF ¬IN.THE.STACK(ARRAYBASE+55) THEN BEGIN
        MOVE.CARD(BASE3+194,BASE3+195);
        LOAD.STACK(ARRAYBASE+55);   END;
        J ← 5;   GO TO SAME;
     L75: IF ¬IN.THE.STACK(PROCBASE+76) THEN BEGIN
        MOVE.CARD(BASE4+3419,BASE4+3428);
        LOAD.STACK(PROCBASE+76);   END;
     IF ¬IN.THE.STACK(PROCBASE+78) THEN BEGIN
        MOVE.CARD(BASE4+3557,BASE4+3576);
        LOAD.STACK(PROCBASE+78);   END;
     IF ¬IN.THE.STACK(PROCBASE+79) THEN BEGIN
        MOVE.CARD(BASE4+3584,BASE4+3613);
        LOAD.STACK(PROCBASE+79);   END;
     IF ¬IN.THE.STACK(PROCBASE+80) THEN BEGIN
        MOVE.CARD(BASE3+196,BASE3+197);
        IF ¬IN.THE.STACK(ARRAYBASE+43) THEN BEGIN
           MOVE.CARD(BASE3+198,BASE3+198);
           LOAD.STACK(ARRAYBASE+43);   END;
        MOVE.CARD(BASE4+3621,BASE4+3675);
        LOAD.STACK(PROCBASE+80);   END;
     MOVE.CARD(BASE4+3091,BASE4+3399);
     END ELSE MOVE.CARD(BASE4+3399,BASE4+3399);
     GO TO L275;
```

```
      P76:   GO TO L275;
      P77:   GO TO L275;
      P78:   GO TO L275;
      P79:   GO TO L275;
      P80:   GO TO L275;
      END;
   PHASE.TWO.ASSEMBLE;
   END;
   BEGIN
   PRINT(<'THE SEGMENT IS STORED',E>);
   END;




   BEGIN
SY SEGMENT 57
AL COMMENT   THE HYDRO COMPILER - PHASE THREE;
      OWN INTEGER I,J,K, ARRAYBASE,PROCBASE,SPECBASE,STORBASE,
      STNO,OPNO,DANO,EXCESS,DATABLOCK,TYPE,TRANSFER,SEGS,
      PAIRS,TEMP1,TEMP2;
      OWN LOGIC TYPEDATA;
      OWN BOOLEAN SKIPOVER,DATAREAD,STORED,HAVEPERIOD,ERROR;
      OWN INTEGER ARRAY PROGRAM[1:500,1:2];
   PROCEDURE PHASE.THREE.TRANSFER;
      BEGIN
      INTEGER UPPERBOUND,LOWORD,HIWORD,FIRSTWORD,LASTWORD,
      WORDNO,READPOINT,WRITEPOINT,READBLOCK,WRITEBLOCK;
      LOGIC ARRAY READBUFFER[1:1621],WRITEBUFFER[1:1600];
      LABEL ERR2;
      LIBRARY PROCEDURE ANDFILE,DISCREAD,DISCWRITE;
      PROCEDURE TRANSFER.CARD (A,B);
         VALUE A,B;
         INTEGER A,B;
         BEGIN
         PROCEDURE TRANSFER.BUFFER;
            BEGIN
            FOR READPOINT←FIRSTWORD STEP 1 UNTIL LASTWORD DO BEGIN
               WRITEPOINT ← WRITEPOINT+1;
               WRITEBUFFER[WRITEPOINT] ← READBUFFER[READPOINT];
               IF WRITEPOINT = 1600 THEN BEGIN
                  DISCWRITE(1600,WRITEBUFFER[1],1,WRITEBLOCK,ERR2);
                  WRITEPOINT ← 0;
                  WRITEBLOCK ← WRITEBLOCK+5;
                  WORDNO ← WORDNO+1600;
                  END;
               END;
            END;
         COMMENT   CODE FOR TRANSFER.CARD;
         LOWORD ← 21*A-20;
         HIWORD ← 21*B;
         IF LOWORD > UPPERBOUND THEN BEGIN
         L300:   IF LOWORD > UPPERBOUND THEN BEGIN
            UPPERBOUND ← UPPERBOUND+1600;
            READBLOCK ← READBLOCK+5;
            GO TO L300;    END;
            DISCREAD(1600,READBUFFER[1],12,READBLOCK-5,ERR2);
            GO TO L302;    END;
        IF LOWORD ¬> UPPERBOUND-1600 THEN BEGIN
        L301:    IF LOWORD ¬> UPPERBOUND-1600 THEN BEGIN
            UPPERBOUND ← UPPERBOUND-1500;
            READBLOCK ← READBLOCK-5;
            GO TO L301;    END;
            DISCREAD(1600,READBUFFER[1],12,READBLOCK-5,ERR2);
            END;
        L302:    IF HIWORD ¬> UPPERBOUND THEN GO TO L303;
            FIRSTWORD ← LOWORD-UPPERBOUND+1600;
            LASTWORD ← 1600;
            TRANSFER.BUFFER;
            DISCREAD(1600,READBUFFER[1],12,READBLOCK,ERR2);
            UPPERBOUND ← UPPERBOUND+1500;
            READBLOCK ← READBLOCK+5;
            LOWORD ← UPPERBOUND-1600+1;
            GO TO L302;
        L303: FIRSTWORD ← LOWORD-UPPERBOUND+1600;
                LASTWORD ← HIWORD-UPPERBOUND+1600;
                TRANSFER.BUFFER;
        END;
COMMENT    CODE FOR THE PHASE.THREE PROCEDURE;
        UPPERBOUND ← 0;
        READBLOCK ← 0;
        WRITEBLOCK ← 0;
        WRITEPOINT ← 0;
        WORDNO ← 0;
        ANDFILE('C135WM01',2,12,READBUFFER[1],ERR2);
        FOR I←1 STEP 1 UNTIL PAIRS DO
            TRANSFER.CARD(PROGRAM[I,1],PROGRAM[I,2]);
        I ← WRITEPOINT;
            FOR WRITEPOINT ← WRITEPOINT+1 STEP 1 UNTIL
            (+(I/320)+1)*320 DO
            WRITEBUFFER[WRITEPOINT] ← '     ';
            WRITEPOINT ← WRITEPOINT-1;
        DISCWRITE(WRITEPOINT,WRITEBUFFER[1],1,WRITEBLOCK,ERR2);
            WRITEBLOCK ← WRITEBLOCK + WRITEPOINT/320;
            WORDNO ← WORDNO + WRITEPOINT;
        J ← +((WORDNO/21-+(WORDNO/21))*21+0.1);
            FOR I←1 STEP 1 UNTIL 21-J DO
            READBUFFER[I] ← '     ';
            READBLOCK ← 1000;
        L310: DISCREAD(1600,READBUFFER[I],1,READBLOCK,ERR2);
                DISCWRITE(1600,READBUFFER[1],1,WRITEBLOCK,ERR2);
            READBLOCK ← READBLOCK+5;
            WRITEBLOCK ← WRITEBLOCK+5;
            FOR I←1 STEP 1 UNTIL 21-J DO
            READBUFFER[I] ← READBUFFER[I+1600];
            IF DATABLOCK > READBLOCK THEN GO TO L310;
            DISCWRITE(320,READBUFFER[1],1,WRITEBLOCK,ERR2);
            PRINT(<P>);
COMMENT  RUN THE ASSEMBLED ALGOL PROGRAM;
```

```
WH              CLS  0   1
                TRM      115
                TRA      A+2
                NAM      ALGOL
                CAL      A-1
                TRM      123
AL     ERR2:    PRINT(<'PHASE THREE COMPILER ERROR',E>);
       END;
   PHASE.THREE.TRANSFER;
   END;
   BEGIN
   PRINT(<'THE SEGMENT IS STORED',E>);
   END;
```

SEVEN HYDRO DATA READ PROCEDURES

```
AL BEGIN
   COMMENT   THE HYDRO LANGUAGE - DATA READ PROCEDURES;
   COMMENT   GLOBALS FOR THE DATA READ PROCEDURES;
      INTEGER II,JJ,SUB1,SUB2,SUB3,COLM,EXPO,TEMP;
      LOGIC ABCD;
      LOGIC ARRAY CHAR[1:80], ALPH[1:20];
      BOOLEAN NEGATIVE,ASTERISK,DEFINED,FINISHED;
   COMMENT   DECLARATION OF THE SEVEN DATA READ PROCEDURES;
   PROCEDURE NEXT.COLUMN;
      BEGIN
      COLM ← COLM+1;
         IF COLM > 80 THEN BEGIN
         NAME(II→80(CHAR[II]));
         READ(<E>, →80<1A>);
         COLM ← 1;
         END;
      END;
   REAL PROCEDURE NUMERICAL.DATA;
      BEGIN
      COLM ← COLM-1;
      NEGATIVE ← FALSE;
      NUMERICAL.DATA ← 0;
      EXPO ← 1;
      LLL1:   NEXT.COLUMN;
         IF CHAR[COLM] = ' ' THEN GO TO LLL1;
         IF CHAR[COLM] = '+' THEN GO TO LLL1;
         IF CHAR[COLM] = '-' THEN BEGIN
         NEGATIVE ← TRUE;   GO TO LLL1;   END;
         IF CHAR[COLM] = '0' THEN TEMP ← 0;
         IF CHAR[COLM] = '1' THEN TEMP ← 1;
         IF CHAR[COLM] = '2' THEN TEMP ← 2;
         IF CHAR[COLM] = '3' THEN TEMP ← 3;
         IF CHAR[COLM] = '4' THEN TEMP ← 4;
         IF CHAR[COLM] = '5' THEN TEMP ← 5;
         IF CHAR[COLM] = '6' THEN TEMP ← 6;
         IF CHAR[COLM] = '7' THEN TEMP ← 7;
         IF CHAR[COLM] = '8' THEN TEMP ← 8;
         IF CHAR[COLM] = '9' THEN TEMP ← 9;
         IF CHAR[COLM] = '.' THEN GO TO LLL2;
         IF CHAR[COLM] = ',' ∨ CHAR[COLM] = ';' ∨
            CHAR[COLM] = ')' THEN GO TO LLL3;
      NUMERICAL.DATA ← 10*NUMERICAL.DATA + TEMP;
      IF EXPO < 1 THEN EXPO ← EXPO-1;
      GO TO LLL1;
```

```
A3.2
      LLL2:   EXPO ← 0;
      GO TO LLL1;
      LLL3:   IF EXPO < 0 THEN
      NUMERICAL.DATA ← NUMERICAL.DATA * 1n*EXPO;
      IF NEGATIVE THEN NUMERICAL.DATA ← -NUMERICAL.DATA;
      FOR JJ ← 2 STEP 1 UNTIL 2n DO
      ALPH[JJ] ← '    ';
      END;
   PROCEDURE ALPHA.NUMERICAL.DATA;
      BEGIN
      FOR JJ ← 1 STEP 1 UNTIL 2n DO
      ALPH[JJ] ← '    ';
      ASTERISK ← FALSE;
      II ← 1;
      LLL4:  FOR JJ ← 1 STEP 1 UNTIL 4 DO BEGIN
         ALPH[II] ← 256*ALPH[II];
         IF ¬ASTERISK THEN BEGIN
            IF CHAR[COLM+JJ] = '*' THEN BEGIN
               ASTERISK ← TRUE;
               COLM ← COLM+JJ;   END   ELSE
            ALPH[II] ← ALPH[II]+CHAR[COLM+JJ];   END;   END;
      IF ¬ASTERISK THEN BEGIN
         COLM ← COLM+4;
         II ← II+1;
         GO TO LLL4;   END;
      LLL5:  NEXT.COLUMN;
      IF CHAR[COLM] ≠ ';' ∧ CHAR[COLM] ≠ ',' THEN GO TO LLL5;
      END;
   PROCEDURE DEFINE.SUBSCRIPTS;
      BEGIN
      DEFINED ← FALSE;
      LLL6:  NEXT.COLUMN;
      IF CHAR[COLM] ≠ '←' ∧ CHAR[COLM] ≠ '[' THEN GO TO LLL6;
      IF CHAR[COLM] = '[' THEN BEGIN
         NEXT.COLUMN;
         SUB1 ← NUMERICAL.DATA;
         IF CHAR[COLM] = ',' THEN BEGIN
         NEXT.COLUMN;
         SUB2 ← NUMERICAL.DATA;
         IF CHAR[COLM] = ',' THEN BEGIN
         NEXT.COLUMN;
         SUB3 ← NUMERICAL.DATA;
         END;   END;
         DEFINED ← TRUE;
         GO TO LLL5;
         END ELSE IF ¬ DEFINED THEN
      SUB1 ← SUB2 ← SUB3 ← 1;
      END;
   REAL PROCEDURE DATA.INPUT;
      BEGIN
      FINISHED ← FALSE;
      LLL7:  NEXT.COLUMN;
      IF CHAR[COLM] = ' ' THEN GO TO LLL7;
      IF CHAR[COLM] = '*' THEN ALPHA.NUMERICAL.DATA
      ELSE DATA.INPUT ← NUMERICAL.DATA;
   IF CHAR[COLM] = ';' THEN BEGIN
      FINISHED ← TRUE;
      LLL8:  NEXT.COLUMN;
      IF CHAR[COLM] = ' ' THEN GO TO LLL8;
      COLM ← COLM-1;
      NAME(ABCD);
      READ(<1C>,←COLM<1B>,<4A>);
      IF ALPH[1] = '$$$$' ∨ SUB1 = 1 THEN
      DEFINE.SUBSCRIPTS;   END;
   END;
PROCEDURE SINGLE.READ(VAR1,VAR2,VAR3,TYPE);
      VALUE TYPE;
      INTEGER TYPE;
      REAL VAR1;
      INTEGER VAR2;
      LOGIC VAR3;
   BEGIN
   IF TYPE=1 THEN VAR1 ← DATA.INPUT;
   IF TYPE=2 THEN VAR2 ← DATA.INPUT;
   IF TYPE=3 THEN BEGIN
   VAR3 ← DATA.INPUT;  VAR3 ← ALPH[1];  ALPH[1] ← '$$$$';  END;
   END;
PROCEDURE ARRAY.READ(VAR1,VAR2,VAR3,TYPE,
      FIRST4,BND1,BND2,BND3);
      VALUE TYPE,BND1,BND2,BND3;
      INTEGER TYPE,BND1,BND2,BND3;
      LOGIC FIRST4;
      REAL ARRAY VAR1;
      INTEGER ARRAY VAR2;
      LOGIC ARRAY VAR3;
   BEGIN
   IF ABCD ≠ FIRST4 THEN GO TO ERROR1;
   LLL11:   IF TYPE = 1 THEN BEGIN
      IF BND2 = 0 THEN VAR1[SUB1] ← DATA.INPUT ELSE
      IF BND3 = 0 THEN VAR1[SUB1,SUB2] ← DATA.INPUT ELSE
      VAR1[SUB1,SUB2,SUB3] ← DATA.INPUT;
      GO TO LLL12;   END;
   IF TYPE = 2 THEN BEGIN
      IF BND2 = 0 THEN VAR2[SUB1] ← DATA.INPUT ELSE
      IF BND3 = 0 THEN VAR2[SUB1,SUB2] ← DATA.INPUT ELSE
      VAR2[SUB1,SUB2,SUB3] ← DATA.INPUT;
      GO TO LLL12;   END;
   IF TYPE = 3 THEN BEGIN   INTEGER I,BOUND;
      I ← DATA.INPUT;
      IF BND2 = 0 THEN BOUND ← 1 ELSE
      IF BND3 = 0 THEN BOUND ← BND2 ELSE
      BOUND ← BND3;
      FOR I←1 STEP 1 UNTIL BOUND DO
      IF BND2 = 0 THEN VAR3[SUB1] ← ALPH[I] ELSE
      IF BND3 = 0 THEN VAR3[SUB1,I] ← ALPH[I] ELSE
      VAR3[SUB1,SUB2,I] ← ALPH[I];
      IF ALPH[I] ≠ '    ' THEN GO TO ERROR4;   END;
   LLL12:   IF FINISHED THEN GO TO LLL14;
   IF ((TYPE=1 ∨ TYPE=2) ∧ BND2=0) ∨
      (TYPE=3 ∧ (BND2=0 ∨ BND3=0)) THEN BEGIN
```

```
A3.4        SUB1 ← SUB1+1;
            GO TO LLL13;    END;
         IF ((TYPE=1 ∨ TYPE=2) ∧ BND3=0) ∨
            (TYPE=3 ∧ BND3≠0) THEN BEGIN
            SUB2 ← SUB2+1;
            IF SUB2 > BND2 THEN BEGIN
            SUB2 ← 1;    SUB1 ← SUB1+1;    END;
            GO TO LLL13;    END;
         IF (TYPE=1 ∨ TYPE=2) ∧ BND3≠0 THEN BEGIN
            SUB3 ← SUB3+1;
            IF SUB3 > BND3 THEN BEGIN
            SUB3 ← 1;    SUB2 ← SUB2+1;
            IF SUB2 > BND2 THEN BEGIN
            SUB2 ← 1;    SUB1 ← SUB1+1;    END;    END;    END;
         LLL13:    IF SUB1 > BND1 THEN GO TO ERROR2;
            GO TO LLL11;
         LLL14:    IF TYPE=1 ∨ TYPE=2 THEN BEGIN
            IF ALPH[1] ≠ '$$$$' THEN GO TO ERROR3;
            GO TO LLL15;    END;
            IF TYPE=3 THEN BEGIN
            IF ALPH[1] = '$$$$' THEN GO TO ERROR3;
            ALPH[1] ← '$$$$';
            DEFINE.SUBSCRIPTS;    END;
         LLL15:    END;
      COMMENT   CODE TO INITIALLY DEFINE ABCD AND SUBSCRIPTS;
         COLM ← 80;
         LLL9:  NEXT.COLUMN;
         IF CHAR[COLM] = ' ' THEN GO TO LLL9;
         COLM ← COLM-1;
         NAME(ABCD);
         READ(<1C>,→COLM<1B>,<4A>);
         IF ABCD = 'END;' THEN BEGIN
         COLM ← 80;    GO TO LLL9;    END;
         DEFINE.SUBSCRIPTS;
         ALPH[1] ← '$$$$';
         GO TO LLL10;
```